Vulnerability View

It would be convenient if security problems in software fell neatly into categories that we could dissect and reason about. Unfortunately, almost any reliability bug can also be a security bug — if the circumstances are right. Capturing the core of a risk sometimes requires understanding a broad architectural issue, and sometimes it requires understanding a highly specific detail of coding.

In the CLASP Vulnerability Lexicon, we have attempted to catalog any themes that lead to security problems and to do this at all appropriate levels. As a result, there are a lot of things in the Lexicon that are not often security concerns, or more precisely are only security concerns when some — potentially rare — condition is met.

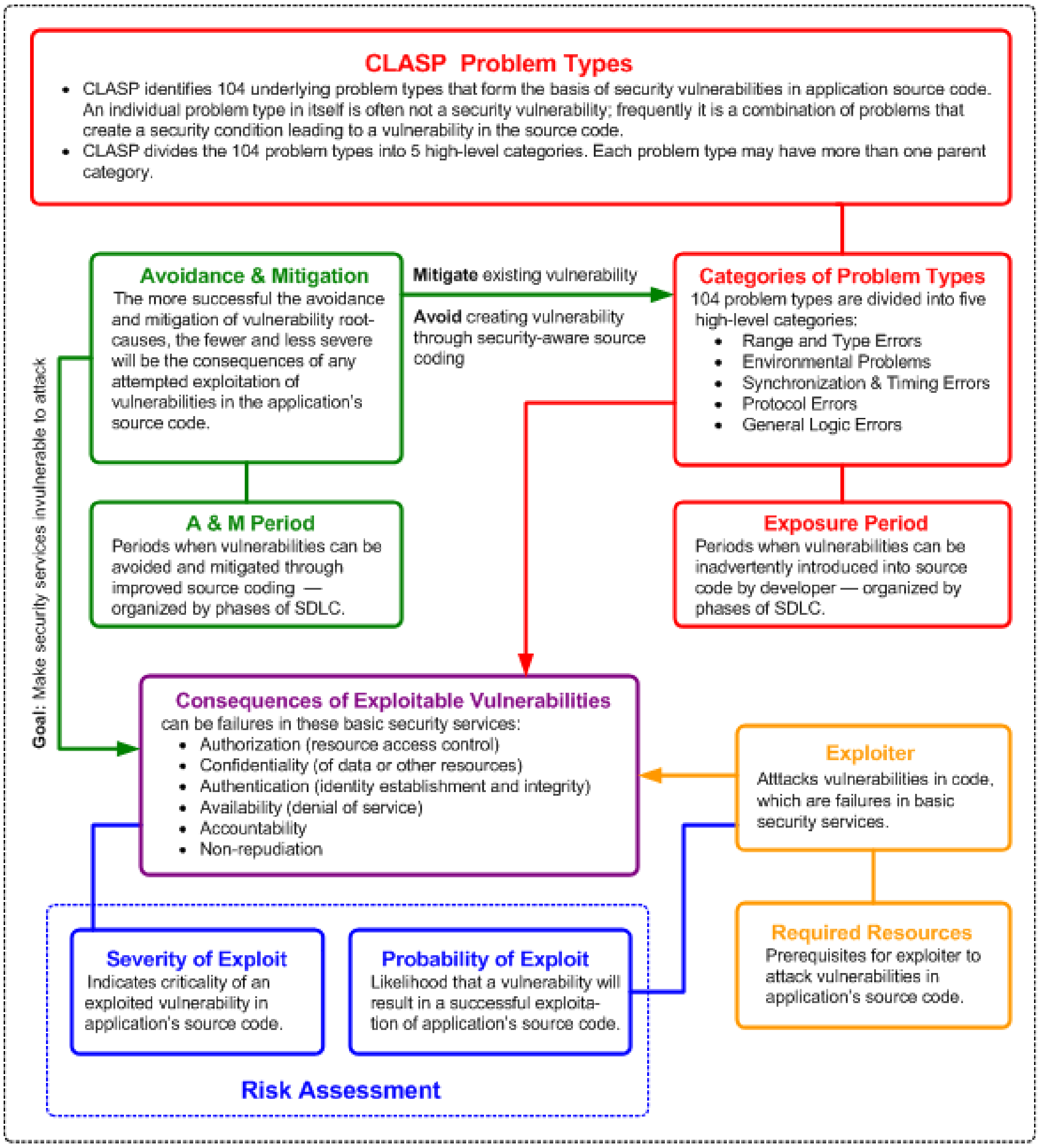
Overview of CLASP Taxonomy

The CLASP taxonomy is a high-level classification of the CLASP process, divided into the following classes for better evaluation and resolution of security vulnerabilities in source code:

- **Problem types** (i.e., basic causes) underlying security-related vulnerabilities.
- **Categories** into which the problem types are divided for diagnostic and resolution purposes.
- **Exposure periods** (i.e., SDLC phases) in which vulnerabilities can be inadvertently introduced into application source code.
- **Consequences** of exploited vulnerabilities for basic security services.
- **Platforms** which may be affected by a vulnerability.
- **Resources** required for attack against vulnerabilities.
- **Risk assessment** of exploitable/exploited vulnerabilities.
- **Avoidance and mitigation periods** (i.e., SDLC phases) in which preventative measures and countermeasures can be applied.

Diagram of Taxonomy

The following figure shows the interaction of the evaluation and resolution classes within the CLASP taxonomy:


CLASP Problem Types
• CLASP identifies 104 underlying problem types that form the basis of security vulnerabilities in application source code. An individual problem type in itself is often not a security vulnerability; frequently it is a combination of problems that create a security condition leading to a vulnerability in the source code.
• CLASP divides the 104 problem types into 5 high-level categories. Each problem type may have more than one parent category.
Avoidance & Mitigation
The more successful the avoidance and mitigation of vulnerability root-causes, the fewer and less severe will be the consequences of any attempted exploitation of vulnerabilities in the application's source code.
Mitigate existing vulnerability
Avoid creating vulnerability through security-aware source coding
Categories of Problem Types
104 problem types are divided into five high-level categories:
• Range and Type Errors
• Environmental Problems
• Synchronization & Timing Errors
• Protocol Errors
• General Logic Errors
A & M Period
Periods when vulnerabilities can be avoided and mitigated through improved source coding — organized by phases of SDLC.
Exposure Period
Periods when vulnerabilities can be inadvertently introduced into source code by developer — organized by phases of SDLC.
Goal: Make security services invulnerable to attack
Consequences of Exploitable Vulnerabilities
can be failures in these basic security services:
• Authorization (resource access control)
• Confidentiality (of data or other resources)
• Authentication (identity establishment and integrity)
• Availability (denial of service)
• Accountability
• Non-repudiation
Exploiter
Attacks vulnerabilities in code, which are failures in basic security services.
Required Resources
Prerequisites for exploiter to attack vulnerabilities in application's source code.
Severity of Exploit
Indicates criticality of an exploited vulnerability in application's source code.
Probability of Exploit
Likelihood that a vulnerability will result in a successful exploitation of application's source code.
Risk Assessment

Classes in CLASP Taxonomy

The CLASP taxonomy is a high-level classification of the CLASP process, divided into classes. These classes enable the software development team to better evaluate and resolve security-related problems. The CLASP classes are:

- CLASP Problem Types
- Categories of Problem Types
- Exposure Periods
- Consequences of Vulnerabilities
- Platforms
- Resources for Attack
- Risk Assessment
- Avoidance and Mitigation Periods
- Other Recorded Information

CLASP Problem Types

We find that individual types of security flaws can — at the highest levels — be introduced for many reasons, including: poor or misunderstood requirements; improper specification; sloppy implementation; flawed components; malicious introduction, etc. Such a breakout — although it is not conducive to organizing software-security problems in an easily understandable way — accurately reflects how, where, and why flaws occur.

Since this taxonomy does not classify individual instances of problems, it really is, to some degree, a catalogue of potential basic causes (or contributing causes).

CLASP identifies 104 underlying problem types — i.e., basic causes — that form the basis of security vulnerabilities in application source code. An individual problem type in itself is often not a security vulnerability; frequently it is a combination of problems that create a security condition leading to a vulnerability in the source code.

Our notion of problem type matches to the notion of “basic cause” — except that we note that individual vulnerabilities are often composed of multiple problems that combine to create a security condition. The individual problems are often not security flaws in and of themselves.

Categories of Problem Types

The problem types in CLASP are individually documented within a very broad set of “categories” but interrelate in a way that is mostly hierarchical. The breakout categories was chosen to be as natural as possible to practitioners in the space, making it somewhat ad hoc. In particular, there are many implicit categories. For example, we define top-level categories, most of which could be considered subcategories of “generic logical flaws,” yet this category does little to advance understanding about actual security issues.

CLASP divides the 104 problem types — i.e., basic causes of vulnerabilities — into five high-level categories. Each problem type may have more than one parent category. These categories are:

- Range and Type Errors
- Environmental Problems
- Synchronization & Timing Errors
- Protocol Errors
- General Logic Errors

These top-level categories each have their own entries. Subcategories (i.e., problem types) are largely hierarchical (i.e., one problem type relates to one “parent” category), although there are some cases where a specific problem type has multiple parents.

Exposure Periods

Another means for evaluating problems is the “exposure period.” In CLASP, exposure period refers to the times in the software development lifecycle when the bug can be introduced into a system. This will generally be one or more of the following: requirements specification; architecture and design; implementation; and deployment.

Failures introduced late in the lifecycle can often be avoided by making different decisions earlier in the lifecycle. For example, deployment problems are generally misconfigurations — and as such can often be explicitly avoided with different up-front decisions.

Consequences of Vulnerabilities

Another class for evaluating problems is the consequence of the flaw. A vulnerability in the source code can lead to a failure of a security service. This is a high-level view of the security services that can fail due to vulnerabilities in source code:

- Authorization (resource access control)
- Confidentiality (of data or other resources)
- Authentication (identity establishment and integrity)
- Availability (denial of service)
- Accountability
- Non-repudiation

This is a more structured way of thinking about security issues than typically used. For example, buffer overflow conditions are usually availability problems because they tend to cause crashes, but often an attacker can escalate privileges or otherwise perform operations under a given privilege that were implicitly not allowed (e.g., overwriting sensitive data), which is ultimately a failure in authorization. In many circumstances, the failure in authorization may be used to thwart other security services, but that is not the direct consequence.

Whether a problem is considered “real” or exploitable is dependent on a security policy that is often implicit. For example, users might consider a system that leaks their personal data to be broken (a lack of privacy, a confidentiality failure). Yet the system designer may not consider this an issue. When evaluating a system, the evaluator should consider the specified requirements and also consider likely implicit requirements of the system users.

Similarly, an important aspect to evaluate about the consequence is “severity.” While we give some indication of Severity ranges, the ultimate determination can only be made on the basis of a set of requirements — and different participants may have different requirements.

Platform

An indication of what platforms may be affected. Here, we use the term in a broad sense. It may mean programming language (e.g., some vulnerabilities common in C and C++ are not possible in other languages), or it may mean operating system, etc.

Resources for Attack

Which resources must the attacker have to exploit an issue? For example, does the attack require local access to the machine running the application? This information can be used to determine whether a particular risk may apply to a given system.

Risk Assessment

There are two categories under Risk Assessment:

- Severity — A relative indication of how critical the problem tends to be in a system, when exploitable.
- Likelihood of exploit — If a particular problem exists in code, what is the likelihood that it will result in an exploitable security condition, given common system requirements?

Avoidance and Mitigation Periods

We provide a high-level overview of some of the more important techniques — and the SDLC periods where they can occur — for avoiding or mitigating a problem, broken down by where in the development lifecycle the technique is generally applied.

Further Recorded Information

CLASP currently records the following additional information about vulnerability classes:

- **Overview** — A brief summary of the problem.
- **Discussion** — A discussion of key points that can help understand the issue.
- **Examples** — For many problems, we give simple examples to better illustrate the problem. We also try to note real-world instances of the vulnerability (i.e., real software that has fallen victim to the problem).

- **Related problems** — Beyond the obvious, sometimes multiple entries refer to the same basic kind of problem but are specific instances. For example, “buffer overflow” gets its own entry, but we also have entries for many specific kinds of buffer overflow that are subject to different exploitation techniques (e.g., heap overflow and stack overflow), and we have entries for many reliability problems that can cause a logic error resulting in a buffer overflow.

Category 1: Range & Type Errors

This section introduces the vulnerability Problem Types organized under the problem type “range and type errors.“

Buffer overflow

Overview

A buffer overflow condition exists when a program attempts to put more data in a buffer than it can hold or when a program attempts to put data in a memory area past a buffer. In this case, a buffer is a sequential section of memory allocated to contain anything from a character string to an array of integers.

Consequences

- Availability: Buffer overflows generally lead to crashes. Other attacks leading to lack of availability are possible, including putting the program into an infinite loop.
- Access control (instruction processing): Buffer overflows often can be used to execute arbitrary code, which is usually outside the scope of a program’s implicit security policy.
- Other: When the consequence is arbitrary code execution, this can often be used to subvert any other security service.

Exposure period

- Requirements specification: The choice could be made to use a language that is not susceptible to these issues.
- Design: Mitigating technologies such as safe-string libraries and container abstractions could be introduced.
- Implementation: Many logic errors can lead to this condition. It can be exacerbated by lack of or misuse of mitigating technologies.

Platform

- Languages: C, C++, Fortran, Assembly
- Operating platforms: All, although partial preventative measures may be deployed, depending on environment.

Required resources

Any

Severity

Very High

Likelihood of exploit

High to Very High

Avoidance and mitigation

- Pre-design: Use a language or compiler that performs automatic bounds checking.
- Design: Use an abstraction library to abstract away risky APIs. Not a complete solution.
- Pre-design through Build: Compiler-based canary mechanisms such as StackGuard, ProPolice and the Microsoft Visual Studio / GS flag. Unless this provides automatic bounds checking, it is not a complete solution.
- Operational: Use OS-level preventative functionality. Not a complete solution.

Discussion

Buffer overflows are one of the best known types of security problem. The best solution is enforced run-time bounds checking of array access, but many C/C++ programmers assume this is too costly or do not have the technology available to them. Even this problem only addresses failures in access control — as an out-of-bounds access is still an exception condition and can lead to an availability problem if not addressed.

Some platforms are introducing mitigating technologies at the compiler or OS level. All such technologies to date address only a subset of buffer overflow problems and rarely provide complete protection against even that subset. It is more common to make the workload of an attacker much higher — for example, by leaving the attacker to guess an unknown value that changes every program execution.

Examples

There are many real-world examples of buffer overflows, including many popular “industrial” applications, such as e-mail servers (Sendmail) and web servers (Microsoft IIS Server).

In code, here is a simple, if contrived example:

```
void example(char *s) {
  char buf[1024];
  strcpy(buf, s);
}
int main(int argc, char **argv) {
  example(argv[1]);
}
```

Since argv[1] can be of any length, more than 1024 characters can be copied into the variable buf.

Related problems

- Stack overflow
- Heap overflow
- Integer overflow

“Write-what-where” condition

Overview

Any condition where the attacker has the ability to write an arbitrary value to an arbitrary location, often as the result of a buffer overflow.

Consequences

- Access control (memory and instruction processing): Clearly, write-what-where conditions can be used to write data to areas of memory outside the scope of a policy. Also, they almost invariably can be used to execute arbitrary code, which is usually outside the scope of a program’s implicit security policy.
- Availability: Many memory accesses can lead to program termination, such as when writing to addresses that are invalid for the current process.
- Other: When the consequence is arbitrary code execution, this can often be used to subvert any other security service.

Exposure period

- Requirements: At this stage, one could specify an environment that abstracts memory access, instead of providing a single, flat address space.
- Design: Many write-what-where problems are buffer overflows, and mitigating technologies for this subset of problems can be chosen at this time.
- Implementation: Any number of simple implementation flaws may result in a write-what-where condition.

Platform

- Languages: C, C++, Fortran, Assembly
- Operating platforms: All, although partial preventative measures may be deployed depending on environment.

Required resources

Any

Severity

Very High

Likelihood of exploit

High

Avoidance and mitigation

- Pre-design: Use a language that provides appropriate memory abstractions.
- Design: Integrate technologies that try to prevent the consequences of this problems.

- Implementation: Take note of mitigations provided for other flaws in this taxonomy that lead to write-what-where conditions.
- Operational: Use OS-level preventative functionality integrated after the fact. Not a complete solution.

Discussion

When the attacker has the ability to write arbitrary data to an arbitrary location in memory, the consequences are often arbitrary code execution. If the attacker can overwrite a pointer's worth of memory (usually 32 or 64 bits), he can redirect a function pointer to his own malicious code.

Even when the attacker can only modify a single byte using a write-what-where problem, arbitrary code execution can be possible. Sometimes this is because the same problem can be exploited repeatedly to the same effect. Other times it is because the attacker can overwrite security-critical application-specific data — such as a flag indicating whether the user is an administrator.

Examples

The classic example of a write-what-where condition occurs when the accounting information for memory allocations is overwritten in a particular fashion.

Here is an example of potentially vulnerable code:

```
#define BUFSIZE 256

int main(int argc, char **argv) {
  char *buf1 = (char *) malloc(BUFSIZE);
  char *buf2 = (char *) malloc(BUFSIZE);

  strcpy(buf1, argv[1]);
  free(buf2);
}
```

Vulnerability in this case is dependent on memory layout. The call to strcpy() can be used to write past the end of buf1, and, with a typical layout, can overwrite the accounting information that the system keeps for buf2 when it is allocated. This information is usually kept before the allocated memory. Note that — if the allocation header for buf2 can be overwritten — buf2 itself can be overwritten as well.

The allocation header will generally keep a linked list of memory "chunks". Particularly, there may be a "previous" chunk and a "next" chunk. Here, the previous chunk for buf2 will probably be buf1, and the next chunk may be null. When the free() occurs, most memory allocators will rewrite the linked list using data from buf2. Particularly, the "next" chunk for buf1 will be updated and the "previous" chunk for any subsequent chunk will be updated. The attacker can insert a memory address for the "next" chunk and a value to write into that memory address for the "previous" chunk.

This could be used to overwrite a function pointer that gets dereferenced later, replacing it with a memory address that the attacker has legitimate access to, where he has placed malicious code, resulting in arbitrary code execution.

There are some significant restrictions that will generally apply to avoid causing a crash in updating headers, but this kind of condition generally results in an exploit.

Related problems

- Buffer overflow
- Format string vulnerabilities

Stack overflow

Overview

A stack overflow condition is a buffer overflow condition, where the buffer being overwritten is allocated on the stack (i.e., is a local variable or, rarely, a parameter to a function).

Consequences

- Availability: Buffer overflows generally lead to crashes. Other attacks leading to lack of availability are possible, including putting the program into an infinite loop.
- Access control (memory and instruction processing): Buffer overflows often can be used to execute arbitrary code, which is usually outside the scope of a program’s implicit security policy.
- Other: When the consequence is arbitrary code execution, this can often be used to subvert any other security service.

Exposure period

- Requirements specification: The choice could be made to use a language that is not susceptible to these issues.
- Design: Mitigating technologies such as safe string libraries and container abstractions could be introduced.
- Implementation: Many logic errors can lead to this condition. It can be exacerbated by lack of or misuse of mitigating technologies.

Platform

- Languages: C, C++, Fortran, Assembly
- Operating platforms: All, although partial preventative measures may be deployed depending on environment.

Required resources

Any

Severity

Very high

Likelihood of exploit

Very high

Avoidance and mitigation

- Pre-design: Use a language or compiler that performs automatic bounds checking.
- Design: Use an abstraction library to abstract away risky APIs. Not a complete solution.
- Pre-design through Build: Compiler-based canary mechanisms such as StackGuard, ProPolice and the Microsoft Visual Studio / GS flag. Unless this provides automatic bounds checking, it is not a complete solution.

- Operational: Use OS-level preventative functionality. Not a complete solution.

Discussion

There are generally several security-critical data on an execution stack that can lead to arbitrary code execution. The most prominent is the stored return address, the memory address at which execution should continue once the current function is finished executing. The attacker can overwrite this value with some memory address to which the attacker also has write access, into which he places arbitrary code to be run with the full privileges of the vulnerable program.

Alternately, the attacker can supply the address of an important call, for instance the POSIX system() call, leaving arguments to the call on the stack. This is often called a *return into libc* exploit, since the attacker generally forces the program to jump at return time into an interesting routine in the C standard library (libc).

Other important data commonly on the stack include the stack pointer and frame pointer, two values that indicate offsets for computing memory addresses. Modifying those values can often be leveraged into a “write-what-where” condition.

Examples

While the buffer overflow example above counts as a stack overflow, it is possible to have even simpler, yet still exploitable, stack based buffer overflows:

```
#define BUFSIZE 256

int main(int argc, char **argv) {
  char buf[BUFSIZE];

  strcpy(buf, argv[1]);
}
```

Related problems

- Parent categories: Buffer overflow
- Subcategories: return address overwrite, stack pointer overwrite, frame pointer overwrite.
- Can be: Function pointer overwrite, array indexer overwrite, write-what-where condition, etc.

Heap overflow

Overview

A heap overflow condition is a buffer overflow, where the buffer that can be overwritten is allocated in the heap portion of memory, generally meaning that the buffer was allocated using a routine such as the POSIX malloc() call.

Consequences

- Availability: Buffer overflows generally lead to crashes. Other attacks leading to lack of availability are possible, including putting the program into an infinite loop.
- Access control (memory and instruction processing): Buffer overflows often can be used to execute arbitrary code, which is usually outside the scope of a program's implicit security policy.
- Other: When the consequence is arbitrary code execution, this can often be used to subvert any other security service.

Exposure period

- Requirements specification: The choice could be made to use a language that is not susceptible to these issues.
- Design: Mitigating technologies such as safe string libraries and container abstractions could be introduced.
- Implementation: Many logic errors can lead to this condition. It can be exacerbated by lack of or misuse of mitigating technologies.

Platform

- Languages: C, C++, Fortran, Assembly
- Operating platforms: All, although partial preventative measures may be deployed depending on environment.

Required resources

Any

Severity

Very High

Likelihood of exploit

- Availability: Very High
- Access control (instruction processing): High

Avoidance and mitigation

- Pre-design: Use a language or compiler that performs automatic bounds checking.
- Design: Use an abstraction library to abstract away risky APIs. Not a complete solution.

- Pre-design through Build: Canary style bounds checking, library changes which ensure the validity of chunk data, and other such fixes are possible, but should not be relied upon.
- Operational: Use OS-level preventative functionality. Not a complete solution.

Discussion

Heap overflows are usually just as dangerous as stack overflows. Besides important user data, heap overflows can be used to overwrite function pointers that may be living in memory, pointing it to the attacker’s code.

Even in applications that do not explicitly use function pointers, the run-time will usually leave many in memory. For example, object methods in C++ are generally implemented using function pointers. Even in C programs, there is often a global offset table used by the underlying runtime.

Examples

While the buffer overflow example above counts as a stack overflow, it is possible to have even simpler, yet still exploitable, stack-based buffer overflows:

```
#define BUFSIZE 256

int main(int argc, char **argv) {
  char *buf;

  buf = (char *)malloc(BUFSIZE);
  strcpy(buf, argv[1]);
}
```

Related problems

- Write-what-where

Buffer underwrite

Overview

A buffer underwrite condition occurs when a buffer is indexed with a negative number, or pointer arithmetic with a negative value results in a position before the beginning of the valid memory location.

Consequences

- Availability: Buffer underwrites will very likely result in the corruption of relevant memory, and perhaps instructions, leading to a crash.
- Access Control (memory and instruction processing): If the memory corrupted memory can be effectively controlled, it may be possible to execute arbitrary code. If the memory corrupted is data rather than instructions, the system will continue to function with improper changes, ones made in violation of a policy, whether explicit or implicit.
- Other: When the consequence is arbitrary code execution, this can often be used to subvert any other security service.

Exposure period

- Requirements specification: The choice could be made to use a language that is not susceptible to these issues.
- Implementation: Many logic errors can lead to this condition. It can be exacerbated by lack of or misuse of mitigating technologies.

Platform

- Languages: C, C++, Assembly
- Operating Platforms: All

Required resources

Any

Severity

High

Likelihood of exploit

Medium

Avoidance and mitigation

- Requirements specification: The choice could be made to use a language that is not susceptible to these issues.
- Implementation: Sanity checks should be performed on all calculated values used as index or for pointer arithmetic.

Examples

The following is an example of code that may result in a buffer underwrite, should find() returns a negative value to indicate that *ch* is not found in srcBuf:

```
int main() {
  ...
  strncpy(destBuf, &srcBuf[find(srcBuf, ch)], 1024);
  ...
}
```

If the index to srcBuf is somehow under user control, this is an arbitrary write-what-where condition.

Related problems

- Buffer Overflow (and related issues)
- Integer Overflow
- Signed-to-unsigned Conversion Error
- Unchecked Array Indexing

Wrap-around error

Overview

Wrap around errors occur whenever a value is incriminated past the maximum value for its type and therefore “wraps around” to a very small, negative, or undefined value.

Consequences

- Availability: Wrap-around errors generally lead to undefined behavior, infinite loops, and therefore crashes.
- Integrity: If the value in question is important to data (as opposed to flow), simple data corruption has occurred. Also, if the wrap around results in other conditions such as buffer overflows, further memory corruption may occur.
- Access control (instruction processing): A wrap around can sometimes trigger buffer overflows which can be used to execute arbitrary code. This is usually outside the scope of a program’s implicit security policy.

Exposure period

- Requirements specification: The choice could be made to use a language that is not susceptible to these issues.
- Design: If the flow of the system, or the protocols used, are not well defined, it may make the possibility of wrap-around errors more likely.
- Implementation: Many logic errors can lead to this condition.

Platform

- Language: C, C++, Fortran, Assembly
- Operating System: Any

Required resources

Any

Severity

High

Likelihood of exploit

Medium

Avoidance and mitigation

- Requirements specification: The choice could be made to use a language that is not susceptible to these issues.
- Design: Provide clear upper and lower bounds on the scale of any protocols designed.
- Implementation: Place sanity checks on all incremented variables to ensure that they remain within reasonable bounds.

Discussion

Due to how addition is performed by computers, if a primitive is incremented past the maximum value possible for its storage space, the system will fail to recognize this, and therefore increment each bit as if it still had extra space.

Because of how negative numbers are represented in binary, primitives interpreted as signed may “wrap” to very large negative values.

Examples

See the Examples section of the problem type *Integer overflow* for an example of wrap-around errors.

Related problems

- Integer overflow
- Unchecked array indexing

Integer overflow

Overview

An integer overflow condition exists when an integer, which has not been properly sanity checked is used in the determination of an offset or size for memory allocation, copying, concatenation, or similarly. If the integer in question is incremented past the maximum possible value, it may wrap to become a very small, or negative number, therefore providing a very incorrect value.

Consequences

- Availability: Integer overflows generally lead to undefined behavior and therefore crashes. In the case of overflows involving loop index variables, the likelihood of infinite loops is also high.
- Integrity: If the value in question is important to data (as opposed to flow), simple data corruption has occurred. Also, if the integer overflow has resulted in a buffer overflow condition, data corruption will most likely take place.
- Access control (instruction processing): Integer overflows can sometimes trigger buffer overflows which can be used to execute arbitrary code. This is usually outside the scope of a program’s implicit security policy.

Exposure period

- Requirements specification: The choice could be made to use a language that is not susceptible to these issues.
- Design: Mitigating technologies such as safe string libraries and container abstractions could be introduced. (This will only prevent the transition from integer overflow to buffer overflow, and only in some cases.)
- Implementation: Many logic errors can lead to this condition. It can be exacerbated by lack of or misuse of mitigating technologies.

Platform

- Languages: C, C++, Fortran, Assembly
- Operating platforms: All

Required resources

Any

Severity

High

Likelihood of exploit

Medium

Avoidance and mitigation

- Pre-design: Use a language or compiler that performs automatic bounds checking.

- Design: Use of sanity checks and assertions at the object level. Ensure that all protocols are strictly defined, such that all out of bounds behavior can be identified simply.
- Pre-design through Build: Canary style bounds checking, library changes which ensure the validity of chunk data, and other such fixes are possible but should not be relied upon.

Discussion

Integer overflows are for the most part only problematic in that they lead to issues of availability. Common instances of this can be found when primitives subject to overflow are used as a loop index variable.

In some situations, however, it is possible that an integer overflow may lead to an exploitable buffer overflow condition. In these circumstances, it may be possible for the attacker to control the size of the buffer as well as the execution of the program.

Recently, a number of integer overflow-based, buffer-overflow conditions have surfaced in prominent software packages. Due to this fact, the relatively difficult to exploit condition is now more well known and therefore more likely to be attacked. The best strategy for mitigation includes: a multi-level strategy including the strict definition of proper behavior (to restrict scale, and therefore prevent integer overflows long before they occur); frequent sanity checks; preferably at the object level; and standard buffer overflow mitigation techniques.

Examples

Integer overflows can be complicated and difficult to detect. The following example is an attempt to show how an integer overflow may lead to undefined looping behavior:

```
short int bytesRec = 0;
char buf[SOMEBIGNUM];

while(bytesRec < MAXGET) {
  bytesRec += getFromInput(buf+bytesRec);
}
```

In the above case, it is entirely possible that bytesRec may overflow, continuously creating a lower number than MAXGET and also overwriting the first MAXGET-1 bytes of buf.

Related problems

- Buffer overflow (and related vulnerabilities): Integer overflows are often exploited only by creating buffer overflow conditions to take advantage of.

Integer coercion error

Overview

Integer coercion refers to a set of flaws pertaining to the type casting, extension, or truncation of primitive data types.

Consequences

- Availability: Integer coercion often leads to undefined states of execution resulting in infinite loops or crashes.
- Access Control: In some cases, integer coercion errors can lead to exploitable buffer overflow conditions, resulting in the execution of arbitrary code.
- Integrity: Integer coercion errors result in an incorrect value being stored for the variable in question.

Exposure period

- Requirements specification: A language which throws exceptions on ambiguous data casts might be chosen.
- Design: Unnecessary casts are brought about through poor design of function interaction
- Implementation: Lack of knowledge on the effects of data casts is the primary cause of this flaw

Platform

- Language: C, C++, Assembly
- Platform: All

Required resources

Any

Severity

High

Likelihood of exploit

Medium

Avoidance and mitigation

- Requirements specification: A language which throws exceptions on ambiguous data casts might be chosen.
- Design: Design objects and program flow such that multiple or complex casts are unnecessary
- Implementation: Ensure that any data type casting that you must used is entirely understood in order to reduce the plausibility of error in use.

Discussion

Several flaws fall under the category of integer coercion errors. For the most part, these errors in and of themselves result only in availability and data integrity issues. However, in some circumstances, they may result in other, more complicated security related flaws, such as buffer overflow conditions.

Examples

See the Examples section of the problem type *Unsigned to signed conversion error* for an example of integer coercion errors.

Related problems

- Signed to unsigned conversion error
- Unsigned to signed conversion error
- Truncation error
- Sign-extension error

Truncation error

Overview

Truncation errors occur when a primitive is cast to a primitive of a smaller size and data is lost in the conversion.

Consequences

- Integrity: The true value of the data is lost and corrupted data is used.

Exposure period

- Implementation: Truncation errors almost exclusively occur at implementation time.

Platform

- Languages: C, C++, Assembly
- Operating platforms: All

Required resources

Any

Severity

Low

Likelihood   of exploit

Low

Avoidance and mitigation

- Implementation: Ensure that no casts, implicit or explicit, take place that move from a larger size primitive or a smaller size primitive.

Discussion

When a primitive is cast to a smaller primitive, the high order bits of the large value are lost in the conversion, resulting in a non-sense value with no relation to the original value. This value may be required as an index into a buffer, a loop iterator, or simply necessary state data. In any case, the value cannot be trusted and the system will be in an undefined state.

While this method may be employed viably to isolate the low bits of a value, this usage is rare, and truncation usually implies that an implementation error has occurred.

Examples

This example, while not exploitable, shows the possible mangling of values associated with truncation errors:

```
#include <stdio.h>

int main() {
  int     intPrimitive;
  short   shortPrimitive;

  intPrimitive = (int)(~((int)0) ^ (1 << (sizeof(int)*8-1)));
  shortPrimitive = intPrimitive;

  printf("Int MAXINT: %d\nShort MAXINT: %d\n",
         intPrimitive, shortPrimitive);
  return (0);
}
```

The above code, when compiled and run, returns the following output:

```
Int MAXINT: 2147483647
Short MAXINT: -1
```

A frequent paradigm for such a problem being exploitable is when the truncated value is used as an array index, which can happen implicitly when 64-bit values are used as indexes, as they are truncated to 32 bits.

Related problems

- Signed to unsigned conversion error
- Unsigned to signed conversion error
- Integer coercion error
- Sign extension error

Sign extension error

Overview

If one extends a signed number incorrectly, if negative numbers are used, an incorrect extension may result.

Consequences

- Integrity: If one attempts to sign extend a negative variable with an unsigned extension algorithm, it will produce an incorrect result.
- Authorization: Sign extension errors — if they are used to collect information from smaller signed sources — can often create buffer overflows and other memory based problems.

Exposure period

- Requirements section: The choice to use a language which provides a framework to deal with this could be used.
- Implementation: A logical flaw of this kind might lead to any number of other flaws.

Platform

- Languages: C or C++
- Operating platforms: Any

Required resources

Any

Severity

High

Likelihood of exploit

High

Avoidance and mitigation

- Implementation: Use a sign extension library or standard function to extend signed numbers.
- Implementation: When extending signed numbers fill in the new bits with 0 if the sign bit is 0 or fill the new bits with 1 if the sign bit is 1.

Discussion

Sign extension errors — if they are used to collect information from smaller signed sources — can often create buffer overflows and other memory based problems.

Examples

In C:

```
struct fakeint {
  short f0;
  short zeros;
};
struct fakeint strange;
struct fakeint strange2;

strange.f0=-240;
strange2.f0=240;

strange2.zeros=0;
strange.zeros=0;

printf("%d %d\n",strange.f0,strange);
printf("%d %d\n",strange2.f0,strange2);
```

Related problems

Not available.

Signed to unsigned conversion error

Overview

A signed-to-unsigned conversion error takes place when a signed primitive is used as an unsigned value, usually as a size variable.

Consequences

- Availability: Incorrect sign conversions generally lead to undefined behavior, and therefore crashes.
- Integrity: If a poor cast lead to a buffer overflow or similar condition, data integrity may be affected.
- Access control (instruction processing): Improper signed-to-unsigned conversions without proper checking can sometimes trigger buffer overflows which can be used to execute arbitrary code. This is usually outside the scope of a program's implicit security policy.

Exposure period

- Requirements specification: The choice could be made to use a language that is not susceptible to these issues.
- Design: Accessor functions may be designed to mitigate some of these logical issues.
- Implementation: Many logic errors can lead to this condition. It can be exacerbated by lack, or misuse, of mitigating technologies.

Platform

- Languages: C, C++, Fortran, Assembly
- Operating platforms: All

Required resources

Any

Severity

High

Likelihood of exploit

Medium

Avoidance and mitigation

- Requirements specification: Choose a language which is not subject to these casting flaws.
- Design: Design object accessor functions to implicitly check values for valid sizes. Ensure that all functions which will be used as a size are checked previous to use as a size. If the language permits, throw exceptions rather than using in-band errors.
- Implementation: Error check the return values of all functions. Be aware of implicit casts made, and use unsigned variables for sizes if at all possible.

Discussion

Often, functions will return negative values to indicate a failure state. In the case of functions which return values which are meant to be used as sizes, negative return values can have unexpected results. If these values are passed to the standard memory copy or allocation functions, they will implicitly cast the negative error-indicating value to a large unsigned value.

In the case of allocation, this may not be an issue; however, in the case of memory and string copy functions, this can lead to a buffer overflow condition which may be exploitable.

Also, if the variables in question are used as indexes into a buffer, it may result in a buffer underflow condition.

Examples

In the following example, it is possible to request that memcpy move a much larger segment of memory than assumed:

```
int returnChunkSize(void *) {
  /* if chunk info is valid, return the size of usable memory,
   * else, return -1 to indicate an error
   */
   ....
}

int main() {
  ...
  memcpy(destBuf, srcBuf, (returnChunkSize(destBuf)-1));
  ...
}
```

If returnChunkSize() happens to encounter an error, and returns -1, memcpy will assume that the value is unsigned and therefore interpret it as MAXINT-1, therefore copying far more memory than is likely available in the destination buffer.

Related problems

- Buffer overflow (and related conditions)
- Buffer underwrite

Unsigned to signed conversion error

Overview

An unsigned-to-signed conversion error takes place when a large unsigned primitive is used as an signed value — usually as a size variable.

Consequences

- Availability: Incorrect sign conversions generally lead to undefined behavior, and therefore crashes.
- Integrity: If a poor cast lead to a buffer underwrite, data integrity may be affected.
- Access control (instruction processing): Improper unsigned-to-signed conversions, often create buffer underwrite conditions which can be used to execute arbitrary code. This is usually outside the scope of a program's implicit security policy.

Exposure period

- Requirements specification: The choice could be made to use a language that is not susceptible to these issues.
- Design: Accessor functions may be designed to mitigate some of these logical issues.
- Implementation: Many logic errors can lead to this condition. It can be exacerbated by lack of or misuse of mitigating technologies.

Platform

- Languages: C, C++, Fortran, Assembly
- Operating platforms: All

Required resources

Any

Severity

High

Likelihood   of exploit

Low to Medium

Avoidance and mitigation

- Requirements specification: The choice could be made to use a language that is not susceptible to these issues.
- Design: Ensure that interacting functions retain the same types and that only safe type casts must occur. If possible, use intelligent marshalling routines to translate between objects.
- Implementation: Use out-of-data band channels for transmitting error messages if unsigned size values must be transmitted. Check all errors.
- Build: Pay attention to compiler warnings which may alert you to improper type casting.

Discussion

Although less frequent an issue than signed-to-unsigned casting, unsigned-to-signed casting can be the perfect precursor to dangerous buffer underwrite conditions that allow attackers to move down the stack where they otherwise might not have access in a normal buffer overflow condition.

Buffer underwrites occur frequently when large unsigned values are cast to signed values, and then used as indexes into a buffer or for pointer arithmetic.

Examples

While not exploitable, the following program is an excellent example of how implicit casts, while not changing the value stored, significantly changes its use:

```
#include <stdio.h>

int main() {
  int value;
  value = (int)(~((int)0) ^ (1 << (sizeof(int)*8)));

  printf("Max unsigned int: %u %1$x\nNow signed: %1$d %1$x\n",
         value);
  return (0);
}
The above code produces the following output:
Max unsigned int: 4294967295 ffffffff
Now signed: -1 ffffffff
```

Note how the hex value remains unchanged.

Related problems

- Buffer underwrite

Unchecked array indexing

Overview

Unchecked array indexing occurs when an unchecked value is used as an index into a buffer.

Consequences

- Availability: Unchecked array indexing will very likely result in the corruption of relevant memory and perhaps instructions, leading to a crash, if the values are outside of the valid memory area
- Integrity: If the memory corrupted is data, rather than instructions, the system will continue to function with improper values.
- Access Control: If the memory corrupted memory can be effectively controlled, it may be possible to execute arbitrary code, as with a standard buffer overflow.

Exposure period

- Requirements specification: The choice could be made to use a language that is not susceptible to these issues.
- Implementation: Many logic errors can lead to this condition. It can be exacerbated by lack of or misuse of mitigating technologies.

Platform

- Languages: C, C++, Assembly
- Operating Platforms: All

Required resources

Any

Severity

Medium

Likelihood of exploit

Medium

Avoidance and mitigation

- Requirements specification: The choice could be made to use a language that is not susceptible to these issues.
- Implementation: Include sanity checks to ensure the validity of any values used as index variables. In loops, use greater-than-or-equal-to, or less-than-or-equal-to, as opposed to simply greater-than, or less-than compare statements.

Discussion

Unchecked array indexing, depending on its instantiation, can be responsible for any number of related issues. Most prominent of these possible flaws is the buffer overflow condition. Due to this fact, consequences range from denial of service, and data corruption, to full blown arbitrary code execution

The most common condition situation leading to unchecked array indexing is the use of loop index variables as buffer indexes. If the end condition for the loop is subject to a flaw, the index can grow or shrink unbounded, therefore causing a buffer overflow or underflow.   Another common situation leading to this condition is the use of a function’s return value, or the resulting value of a calculation directly as an index in to a buffer.

Examples

Not available.

Related problems

- Buffer Overflow (and related issues)
- Buffer Underwrite
- Signed-to-Unsigned Conversion Error
- Write-What-Where

Miscalculated null termination

Overview

Miscalculated null termination occurs when the placement of a null character at the end of a buffer of characters (or string) is misplaced or omitted.

Consequences

- Confidentiality: Information disclosure may occur if strings with misplaced or omitted null characters are printed.
- Availability: A randomly placed null character may put the system into an undefined state, and therefore make it prone to crashing.
- Integrity: A misplaced null character may corrupt other data in memory
- Access Control: Should the null character corrupt the process flow, or effect a flag controlling access, it may lead to logical errors which allow for the execution of arbitrary code.

Exposure period

- Requirements specification: The choice could be made to use a language that is not susceptible to these issues.
- Implementation: Precise knowledge of string manipulation functions may prevent this issue

Required resources

Any

Severity

High

Likelihood of exploit

Medium

Avoidance and mitigation

- Requirements specification: The choice could be made to use a language that is not susceptible to these issues.
- Implementation: Ensure that all string functions used are understood fully as to how they append null characters. Also, be wary of off-by-one errors when appending nulls to the end of strings.

Discussion

Miscalculated null termination is a common issue, and often difficult to detect. The most common symptoms occur infrequently (in the case of problems resulting from “safe” string functions), or in odd ways characterized by data corruption (when caused by off-by-one errors).

The case of an omitted null character is the most dangerous of the possible issues. This will almost certainly result in information disclosure, and possibly a buffer overflow condition, which may be exploited to execute arbitrary code.

As for misplaced null characters, the biggest issue is a subset of buffer overflow, and write-what-where conditions, where data corruption occurs from the writing of a null character over valid data, or even instructions. These logic issues may result in any number of security flaws.

Examples

While the following example is not exploitable, it provides a good example of how nulls can be omitted or misplaced, even when “safe” functions are used:

```
#include <stdio.h>
#include <string.h>

int main() {
  char longString[] = "Cellular bananular phone";
  char shortString[16];

  strncpy(shortString, longString, 16);
  printf("The last character in shortString is: %c %1$x\n",
         shortString[15]);
  return (0);
}
```

The above code gives the following output:

```
The last character in shortString is: l 6c
```

So, the shortString array does not end in a NULL character, even though the “safe” string function strncpy() was used.

Related problems

- Buffer overflow (and related issues)
- Write-what-where: A subset of the problem in some cases, in which an attacker may write a null character to a small range of possible addresses.

Improper string length checking

Overview

Improper string length checking takes place when wide or multi-byte character strings are mistaken for standard character strings.

Consequences

- Access control: This flaw is exploited most frequently when it results in a buffer overflow condition, which leads to arbitrary code execution.
- Availability: Even if the flaw remains unexploded, the probability that the process will crash due to the writing of data over arbitrary memory may result in a crash.

Exposure period

- Requirements specification: A language which is not subject to this flaw may be chosen.
- Implementation: Misuse of string functions at implementation time is the most common cause of this problem.
- Build: Compile-time mitigation techniques may serve to complicate exploitation.

Platform

- Language: C, C++, Assembly
- Platform: All

Required resources

Any

Severity

High

Likelihood of exploit

High

Avoidance and mitigation

- Requirements specification: A language which is not subject to this flaw may be chosen.
- Implementation: Ensure that if wide or multi-byte strings are in use that all functions which interact with these strings are wide and multi-byte character compatible, and that the maximum character size is taken into account when memory is allocated.
- Build: Use of canary-style overflow prevention techniques at compile time may serve to complicate exploitation but cannot mitigate it fully; nor will this technique have any effect on process stability. This is not a complete mitigation technique.

Discussion

There are several ways in which improper string length checking may result in an exploitable condition. All of these however involve the introduction of buffer overflow conditions in order to reach an exploitable state.

The first of these issues takes place when the output of a wide or multi-byte character string, string-length function is used as a size for the allocation of memory. While this will result in an output of the number of characters in the string, note that the characters are most likely not a single byte, as they are with standard character strings. So, using the size returned as the size sent to new or malloc and copying the string to this newly allocated memory will result in a buffer overflow.

Another common way these strings are misused involves the mixing of standard string and wide or multi-byte string functions on a single string. Invariably, this mismatched information will result in the creation of a possibly exploitable buffer overflow condition.

Again, if a language subject to these flaws must be used, the most effective mitigation technique is to pay careful attention to the code at implementation time and ensure that these flaws do not occur.

Examples

The following example would be exploitable if any of the commented incorrect malloc calls were used.

```
#include <stdio.h>
#include <strings.h>
#include <wchar.h>

int main() {
  wchar_t wideString[] = L"The spazzy orange tiger jumped ” \
                         “over the tawny jaguar.";
  wchar_t *newString;

  printf("Strlen() output: %d\nWcslen() output: %d\n",
         strlen(wideString), wcslen(wideString));

  /* Very wrong for obvious reasons //
  newString = (wchar_t *) malloc(strlen(wideString));
  */

  /* Wrong because wide characters aren't 1 byte long! //
  newString = (wchar_t *) malloc(wcslen(wideString));
  */

  /* correct! */
  newString = (wchar_t *) malloc(wcslen(wideString) *
                         sizeof(wchar_t));

  /* ... */
}
```

The output from the printf() statement would be:

```
Strlen() output: 0
Wcslen() output: 53
```

Related problems

- Buffer overflow (and related issues)

Covert storage channel

Overview

The existence of a covert storage channel in a communications channel may release information which can be of significant use to attackers.

Consequences

- Confidentiality: Covert storage channels may provide attackers with important information about the system in question.

Exposure period

- Implementation: The existence of data in a covert storage channel is largely a flaw caused by implementers.

Platform

- Languages: All
- Operating platforms: All

Required resources

Network proximity: Some ability to sniff network traffic would be required to capitalize on this flaw.

Severity

Medium

Likelihood of exploit

High

Avoidance and mitigation

- Implementation: Ensure that all reserved fields are set to zero before messages are sent and that no unnecessary information is included.

Discussion

Covert storage channels occur when out-of-band data is stored in messages for the purpose of memory reuse. If these messages or packets are sent with the unnecessary data still contained within, it may tip off malicious listeners as to the process that created the message.

With this information, attackers may learn any number of things, including the hardware platform, operating system, or algorithms used by the sender. This information can be of significant value to the user in launching further attacks.

Examples

An excellent example of covert storage channels in a well known application is the ICMP error message echoing functionality. Due to ambiguities in the ICMP RFC, many IP implementations use the memory within the packet for storage or calculation.

For this reason, certain fields of certain packets — such as ICMP error packets which echo back parts of received messages — may contain flaws or extra information which betrays information about the identity of the target operating system.

This information is then used to build up evidence to decide the environment of the target. This is the first crucial step in determining if a given system is vulnerable to a particular flaw and what changes must be made to malicious code to mount a successful attack.

Related problems

Not available.

Failure to account for default case in switch

Overview

The failure to account for the default case in switch statements may lead to complex logical errors and may aid in other, unexpected security-related conditions.

Consequences

- Undefined: Depending on the logical circumstances involved, any consequences may result: e.g., issues of confidentiality, authentication, authorization, availability, integrity, accountability, or non-repudiation.

Exposure period

- Implementation: This flaw is a simple logic issue, introduced entirely at implementation time.

Platform

- Language: Any
- Platform: Any

Required resources

Any

Severity

Undefined.

Likelihood of exploit

Undefined.

Avoidance and mitigation

- Implementation: Ensure that there are no unaccounted for cases, when adjusting flow or values based on the value of a given variable. In switch statements, this can be accomplished through the use of the default label.

Discussion

This flaw represents a common problem in software development, in which not all possible values for a variable are considered or handled by a given process. Because of this, further decisions are made based on poor information, and cascading failure results.

This cascading failure may result in any number of security issues, and constitutes a significant failure in the system. In the case of switch style statements, the very simple act of creating a default case can mitigate this situation, if done correctly.

Often however, the default cause is used simply to represent an assumed option, as opposed to working as a sanity check. This is poor practice and in some cases is as bad as omitting a default case entirely.

Examples

In general, a safe switch statement has this form:

```
switch (value) {
  case 'A':
    printf("A!\n");
    break;
  case 'B':
    printf("B!\n");
    break;
  default:
    printf("Neither A nor B\n");
}
```

This is because the assumption cannot be made that all possible cases are accounted for. A good practice is to reserve the default case for error handling.

Related problems

- Undefined: A logical flaw of this kind might lead to any number of other flaws.

Null-pointer dereference

Overview

A null-pointer dereference takes place when a pointer with a value of NULL is used as though it pointed to a valid memory area.

Consequences

- Availability: Null-pointer dereferences invariably result in the failure of the process.

Exposure period

- Requirements specification: The choice could be made to use a language that is not susceptible to these issues.
- Implementation: Proper sanity checks at implementation time can serve to prevent null-pointer dereferences

Platform

- Languages: C, C++, Assembly
- Platforms: All

Required resources

Any

Severity

Medium

Likelihood of exploit

Medium

Avoidance and mitigation

- Requirements specification: The choice could be made to use a language that is not susceptible to these issues.
- Implementation: If all pointers that could have been modified are sanity-checked previous to use, nearly all null-pointer dereferences can be prevented.

Discussion

Null-pointer dereferences, while common, can generally be found and corrected in a simply way. They will always result in the crash of the process — unless exception handling (on some platforms) in invoked, and even then, little can be done to salvage the process.

Examples

Null-pointer dereference issue can occur through a number of flaws, including race conditions, and simple programming omissions. While there are no complete fixes aside from contentious programming, the following steps will go a long way to ensure that null-pointer dereferences do not occur.

Before using a pointer, ensure that it is not equal to NULL:

```
if (pointer1 != NULL) {
  /* make use of pointer1 */
  /* ... */
}
```

When freeing pointers, ensure they are not set to NULL, and be sure to set them to NULL once they are freed:

```
if (pointer1 != NULL) {
  free(pointer1);
  pointer1 = NULL;
}
```

If you are working with a multi-threaded or otherwise asynchronous environment, ensure that proper locking APIs are used to lock before the if statement; and unlock when it has finished.

Related problems

- Miscalculated null termination
- State synchronization error

Using freed memory

Overview

The use of heap allocated memory after it has been freed or deleted leads to undefined system behavior and, in many cases, to a write-what-where condition.

Consequences

- Integrity: The use of previously freed memory may corrupt valid data, if the memory area in question has been allocated and used properly elsewhere.
- Availability: If chunk consolidation occur after the use of previously freed data, the process may crash when invalid data is used as chunk information.
- Access Control (instruction processing): If malicious data is entered before chunk consolidation can take place, it may be possible to take advantage of a write-what-where primitive to execute arbitrary code.

Exposure period

- Implementation: Use of previously freed memory errors occur largely at implementation time.

Platform

- Languages: C, C++, Assembly
- Operating Platforms: All

Required resources

Any

Severity

Very High

Likelihood   of exploit

High

Avoidance and mitigation

- Implementation: Ensuring that all pointers are set to NULL, once the memory they point to has been freed, can be effective strategy. The utilization of multiple or complex data structures may lower the usefulness of this strategy.

Discussion

The use of previously freed memory can have any number of adverse consequences — ranging from the corruption of valid data to the execution of arbitrary code, depending on the instantiation and timing of the flaw.

The simplest way data corruption may occur involves the system’s reuse of the freed memory. In this scenario, the memory in question is allocated to another pointer validly at some point after it has been freed. The original pointer to the freed memory is used again and points to somewhere within the new allocation. As the data is changed, it corrupts the validly used memory; this induces undefined behavior in the process.

If the newly allocated data chances to hold a class, in C++ for example, various function pointers may be scattered within the heap data. If one of these function pointers is overwritten with an address to valid shellcode, execution of arbitrary code can be achieved.

Examples

The following example

```
#include <stdio.h>
#include <unistd.h>

#define BUFSIZER1   512
#define BUFSIZER2   ((BUFSIZER1/2) - 8)

int main(int argc, char **argv) {
    char *buf1R1;
    char *buf2R1;
    char *buf2R2;
    char *buf3R2;

    buf1R1 = (char *) malloc(BUFSIZER1);
    buf2R1 = (char *) malloc(BUFSIZER1);

    free(buf2R1);

    buf2R2 = (char *) malloc(BUFSIZER2);
    buf3R2 = (char *) malloc(BUFSIZER2);

    strncpy(buf2R1, argv[1], BUFSIZER1-1);
    free(buf1R1);
    free(buf2R2);
    free(buf3R2);
}
```

Related problems

- Buffer overflow (in particular, heap overflows): The method of exploitation is often the same, as both constitute the unauthorized writing to heap memory.
- Write-what-where condition: The use of previously freed memory can result in a write-what-where in several ways.

Doubly freeing memory

Overview

Freeing or deleting the same memory chunk twice may — when combined with other flaws — result in a write-what-where condition.

Consequences

- Access control: Doubly freeing memory may result in a write-what-where condition, allowing an attacker to execute arbitrary code.

Exposure period

- Requirements specification: A language which handles memory allocation and garbage collection automatically might be chosen.
- Implementation: Double frees are caused most often by lower-level logical errors.

Platform

- Language: C, C++, Assembly
- Operating system: All

Required resources

Any

Severity

High

Likelihood of exploit

Low to Medium

Avoidance and mitigation

- Implementation: Ensure that each allocation is freed only once. After freeing a chunk, set the pointer to NULL to ensure the pointer cannot be freed again. In complicated error conditions, be sure that clean-up routines respect the state of allocation properly. If the language is object oriented, ensure that object destructors delete each chunk of memory only once.

Discussion

Doubly freeing memory can result in roughly the same write-what-where condition that the use of previously freed memory will.

Examples

While contrived, this code should be exploitable on Linux distributions which do not ship with heap-chunk check summing turned on.

```
#include <stdio.h>
#include <unistd.h>
```

```
#define BUFSIZE1    512
#define BUFSIZE2    ((BUFSIZE1/2) - 8)

int main(int argc, char **argv) {
  char *buf1R1;
  char *buf2R1;
  char *buf1R2;

  buf1R1 = (char *) malloc(BUFSIZE2);
  buf2R1 = (char *) malloc(BUFSIZE2);

  free(buf1R1);
  free(buf2R1);

  buf1R2 = (char *) malloc(BUFSIZE1);
  strncpy(buf1R2, argv[1], BUFSIZE1-1);

  free(buf2R1);
  free(buf1R2);
}
```

Related problems

- Using freed memory
- Write-what-where

Invoking untrusted mobile code

Overview

This process will download external source or binaries and execute it.

Consequences

Unspecified.

Exposure period

Implementation: This flaw is a simple logic issue, introduced entirely at implementation time.

Platform

Languages: Java and C++

Operating platform: Any

Required resources

Any

Severity

Medium

Likelihood of exploit

Medium

Avoidance and mitigation

- Implementation: Avoid doing this without proper cryptographic safeguards.

Discussion

This is an unsafe practice and should not be performed unless one can use some type of cryptographic protection to assure that the mobile code has not been altered.

Examples

In Java:

```
URL[] classURLs= new URL[]{new URL(“file:subdir/”)};
URLClassLoader loader = nwe URLClassLoader(classURLs);
Class loadedClass = Class.forName(“loadMe”, true, loader);
```

Related problems

- Cross-site scripting

Cross-site scripting

Overview

Cross-site scripting attacks are an instantiation of injection problems, in which malicious scripts are injected into the otherwise benign and trusted web sites.

Consequences

- Confidentiality: The most common attack performed with cross-site scripting involves the disclosure of information stored in user cookies.
- Access control: In some circumstances it may be possible to run arbitrary code on a victim's computer when cross-site scripting is combined with other flaws

Exposure period

- Implementation: If bulletin-board style functionality is present, cross-site scripting may only be deterred at implementation time.

Platform

- Language: Any
- Platform: All (requires interaction with a web server supporting dynamic content)

Required resources

Any

Severity

Medium

Likelihood of exploit

Medium

Avoidance and mitigation

- Implementation: Use a white-list style parsing routine to ensure that no posted content contains scripting tags.

Discussion

Cross-site scripting attacks can occur wherever an untrusted user has the ability to publish content to a trusted web site. Typically, a malicious user will craft a client-side script, which — when parsed by a web browser — performs some activity (such as sending all site cookies to a given E-mail address).

If the input is unchecked, this script will be loaded and run by each user visiting the web site. Since the site requesting to run the script has access to the cookies in question, the malicious script does also.

There are several other possible attacks, such as running "Active X" controls (under Microsoft Internet Explorer) from sites that a user perceives as trustworthy; cookie theft is however by far the most common.

All of these attacks are easily prevented by ensuring that no script tags — or for good measure, HTML tags at all — are allowed in data to be posted publicly.

Examples

Cross-site scripting attacks may occur anywhere that possibly malicious users are allowed to post unregulated material to a trusted web site for the consumption of other valid users.

The most common example can be found in bulletin-board web sites which provide web based mailing list-style functionality.

Related problems

- Injection problems
- Invoking untrusted mobile code

Format string problem

Overview

Format string problems occur when a user has the ability to control or write completely the format string used to format data in the printf style family of C/C++ functions.

Consequences

- Confidentially: Format string problems allow for information disclosure which can severely simplify exploitation of the program.
- Access Control: Format string problems can result in the execution of arbitrary code.

Exposure period

- Requirements specification: A language might be chosen that is not subject to this issue.
- Implementation: Format string problems are largely introduced at implementation time.
- Build: Several format string problems are discovered by compilers

Platform

- Language: C, C++, Assembly
- Platform: Any

Required resources

Any

Severity

High

Likelihood   of exploit

Very High

Avoidance and mitigation

- Requirements specification: Choose a language which is not subject to this flaw.
- Implementation: Ensure that all format string functions are passed a static string which cannot be controlled by the user and that the proper number of arguments are always sent to that function as well. If at all possible, do not use the %n operator in format strings.
- Build: Heed the warnings of compilers and linkers, since they may alert you to improper usage.

Discussion

Format string problems are a classic C/C++ issue that are now rare due to the ease of discovery. The reason format string vulnerabilities can be exploited is due to the %n operator. The %n operator will write the number of characters, which have been printed by the format string therefore far, to the memory pointed to by its argument.

Through skilled creation of a format string, a malicious user may use values on the stack to create a write-what-where condition. Once this is achieved, he can execute arbitrary code.

Examples

The following example is exploitable, due to the printf() call in the printWrapper() function. Note: The stack buffer was added to make exploitation more simple.

```
#include <stdio.h>

void printWrapper(char *string) {
  printf(string);
}

int main(int argc, char **argv) {
  char buf[5012];
  memcpy(buf, argv[1], 5012);
  printWrapper(argv[1]);
  return (0);
}
```

Related problems

- Injection problem
- Write-what-where

Injection problem (“data” used as something else)

Overview

Injection problems span a wide range of instantiations. The basic form of this flaw involves the injection of control-plane data into the data-plane in order to alter the control flow of the process.

Consequences

- Confidentiality: Many injection attacks involve the disclosure of important information — in terms of both data sensitivity and usefulness in further exploitation
- Authentication: In some cases injectable code controls authentication; this may lead to remote vulnerability
- Access Control: Injection attacks are characterized by the ability to significantly change the flow of a given process, and in some cases, to the execution of arbitrary code.
- Integrity: Data injection attacks lead to loss of data integrity in nearly all cases as the control-plane data injected is always incidental to data recall or writing.
- Accountability: Often the actions performed by injected control code are unlogged.

Exposure period

- Requirements specification: A language might be chosen which is not subject to these issues.
- Implementation: Many logic errors can contribute to these issues.

Platform

- Languages: C, C++, Assembly, SQL
- Platforms: Any

Required resources

Any

Severity

High

Likelihood   of exploit

Very High

Avoidance and mitigation

- Requirements specification: A language might be chosen which is not subject to these issues.
- Implementation: As so many possible implementations of this flaw exist, it is best to simply be aware of the flaw and work to ensure that all control characters entered in data are subject to black-list style parsing.

Discussion

Injection problems encompass a wide variety of issues — all mitigated in very different ways. For this reason, the most effective way to discuss these flaws is to note the distinct features which classify them as injection flaws.

The most important issue to note is that all injection problems share one thing in common — i.e., they allow for the injection of control plane data into the user-controlled data plane. This means that the execution of the process may be altered by sending code in through legitimate data channels, using no other mechanism. While buffer overflows, and many other flaws, involve the use of some further issue to gain execution, injection problems need only for the data to be parsed.

The most classing instantiations of this category of flaw are SQL injection and format string vulnerabilities.

Examples

Injection problems describe a large subset of problems with varied instantiations. For an example of one of these problems, see the section *Format string problem*.

Related problems

- SQL injection
- Format String problem
- Command injection
- Log injection
- Reflection injection

Command injection

Overview

Command injection problems are a subset of injection problem, in which the process is tricked into calling external processes of the attackers choice through the injection of control-plane data into the data plane.

Consequences

- Access control: Command injection allows for the execution of arbitrary commands and code by the attacker.

Exposure period

- Design: It may be possible to find alternate methods for satisfying functional requirements than calling external processes. This is minimal.
- Implementation: Exposure for this issue is limited almost exclusively to implementation time. Any language or platform is subject to this flaw.

Platform

- Language: Any
- Platform: Any

Required resources

Any

Severity

High

Likelihood of exploit

Very High

Avoidance and mitigation

- Design: If at all possible, use library calls rather than external processes to recreate the desired functionality
- Implementation: Ensure that all external commands called from the program are statically created, or — if they must take input from a user — that the input and final line generated are vigorously white-list checked.
- Run time: Run time policy enforcement may be used in a white-list fashion to prevent use of any non-sanctioned commands.

Discussion

Command injection is a common problem with wrapper programs. Often, parts of the command to be run are controllable by the end user. If a malicious user injects a character (such as a semi-colon) that delimits the end of one command and the beginning of another, he may then be able to insert an entirely new and unrelated command to do whatever he pleases.

The most effective way to deter such an attack is to ensure that the input provided by the user adheres to strict rules as to what characters are acceptable. As always, white-list style checking is far preferable to black-list style checking.

Examples

The following code is wrapper around the UNIX command *cat* which prints the contents of a file to standard out. It is also injectable:

```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char **argv) {
  char cat[] = "cat ";
  char *command;
  size_t commandLength;

  commandLength = strlen(cat) + strlen(argv[1]) + 1;
  command = (char *) malloc(commandLength);
  strncpy(command, cat, commandLength);
  strncat(command, argv[1], (commandLength - strlen(cat)) );

  system(command);
  return (0);
}
```

Used normally, the output is simply the contents of the file requested:

```
$ ./catWrapper Story.txt
When last we left our heroes...
```

However, if we add a semicolon and another command to the end of this line, the command is executed by catWrapper with no complaint:

```
$ ./catWrapper Story.txt; ls
When last we left our heroes...
Story.txt               doubFree.c              nullpointer.c
unstosig.c              www*                    a.out*
format.c                strlen.c                useFree*
catWrapper*             misnull.c               strlength.c               useFree.c
commandinjection.c      nodefault.c             trunc.c
writeWhatWhere.c
```

If catWrapper had been set to have a higher privilege level than the standard user, arbitrary commands could be executed with that higher privilege.

Related problems

- Injection problem

Log injection

Overview

Log injection problems are a subset of injection problem, in which invalid entries taken from user input are inserted in logs or audit trails, allowing an attacker to mislead administrators or cover traces of attack. Log injection can also sometimes be used to attack log monitoring systems indirectly by injecting data that monitoring system will misinterpret.

Consequences

- Integrity: Logs susceptible to injection can not be trusted for diagnostic or evidentiary purposes in the event of an attack on other parts of the system.
- Access control: Log injection may allow indirect attacks on systems monitoring the log.

Exposure period

- Design: It may be possible to find alternate methods for satisfying functional requirements than allowing external input to be logged.
- Implementation: Exposure for this issue is limited almost exclusively to implementation time. Any language or platform is subject to this flaw.

Platform

- Language: Any
- Platform: Any

Required resources

Any

Severity

High

Likelihood of exploit

Very High

Avoidance and mitigation

- Design: If at all possible, avoid logging data that came from external inputs.
- Implementation: Ensure that all log entries are statically created, or — if they must record external data — that the input is vigorously white-list checked.
- Run time: Avoid viewing logs with tools that may interpret control characters in the file, such as command-line shells.

Discussion

Log injection attacks can be used to cover up log entries or insert misleading entries. Common attacks on logs include inserting additional entries with fake information, truncating entries to cause information loss, or using control characters to hide entries from certain file viewers.

The most effective way to deter such an attack is to ensure that any external input being logged adheres to strict rules as to what characters are acceptable. As always, white-list style checking is far preferable to black-list style checking.

Examples

The following code is a simple Python snippet which writes a log entry to a file. It does not filter log contents:

```
def log_failed_login(username)
 log = open(“access.log”, ‘a’)
 log.write(“User login failed for: %s\n“ % username)
 log.close()
```

Normal log file output looks like:

```
User login failed for: guest
User login failed for: admin
```

However, if we pass in the following as the username:

```
guest\nUser login succeeded for: admin
```

the log would instead have the misleading entries:

```
User login failed for: guest
User login succeeded for: admin
```

If it was expected that the log was going to be viewed from within a command shell (as is often the case with server software) we could inject terminal control characters to cause the display to back up lines or erase log entries from view. Doing this does not actually remove the entries from the file, but it can prevent casual inspection from noticing security critical log entries.

Related problems

- Injection problem

Reflection injection

Overview

Reflection injection problems are a subset of injection problem, in which external input is used to construct a string value passed to class reflection APIs. By manipulating the value an attacker can cause unexpected classes to be loaded, or change what method or fields are accessed on an object.

Consequences

- Access control: Reflection injection allows for the execution of arbitrary code by the attacker.

Exposure period

- Design: It may be possible to find alternate methods for satisfying functional requirements than using reflection.
- Implementation: Avoid using external input to generate reflection string values.

Platform

- Language: Java, .NET, and other languages that support reflection
- Platform: Any

Required resources

Any

Severity

High

Likelihood of exploit

High

Avoidance and mitigation

- Design: It may be possible to find alternate methods for satisfying functional requirements than using reflection.
- Implementation: Avoid using external input to generate reflection string values.

Discussion

The most straightforward reflection injection attack is to provide the name of an alternate class available to the target application which implements the same interfaces but operates in a less secure manner. This can be used as leverage for more extensive attacks. More complex attacks depend upon the specific deployment situation of the application.

If the classloader being used is capable of remote class fetching this becomes an extremely serious vulnerability, since attackers could supply arbitrary URLs that point at constructed attack classes. In this case, the class doesn't necessarily even need to implement methods that perform the same as the replaced class, since a static initializer could be used to carry out the attack.

If it is necessary to allow reflection utilizing external input, limit the possible values to a predefined list. For example, reflection is commonly used for loading JDBC database connector classes. Most often, the string class name is read from a configuration file. Injection problems can be avoided by embedding a list of strings naming each of the supported database driver classes and requiring the class name read from the file to be in the list before loading.

Examples

The following Java code dynamically loads a connection class to be used for transferring data:

```
// connType is a String read from an external source
Class connClass = Class.forName(connType);
HttpURLConnection conn = (HttpURLConnection)connClass.newInstance();
conn.connect();
```

Suppose this application normally passed “javax.net.ssl.HttpsUrlConnection”. This would provide an HTTPS connection using SSL to protect the transferred data. If an attacker replaced the connType string with “java.net.HttpURLConnection” then all data transfers performed by this code would happened over an un-encrypted HTTP connection instead.

Related problems

- Injection problem

SQL injection

Overview

SQL injection attacks are another instantiation of injection attack, in which SQL commands are injected into data-plane input in order to effect the execution of predefined SQL commands.

Consequences

- Confidentiality: Since SQL databases generally hold sensitive data, loss of confidentiality is a frequent problem with SQL injection vulnerabilities.
- Authentication: If poor SQL commands are used to check user names and passwords, it may be possible to connect to a system as another user with no previous knowledge of the password.
- Authorization: If authorization information is held in an SQL database, it may be possible to change this information through the successful exploitation of an SQL injection vulnerability.
- Integrity: Just as it may be possible to read sensitive information, it is also possible to make changes or even delete this information with an SQL injection attack.

Exposure period

- Requirements specification: A non-SQL style database which is not subject to this flaw may be chosen.
- Implementation: If SQL is used, all flaws resulting in SQL injection problems must be mitigated at the implementation level.

Platform

- Language: SQL
- Platform: Any (requires interaction with an SQL database)

Required resources

Any

Severity

Medium to High

Likelihood of exploit

Very High

Avoidance and mitigation

- Requirements specification: A non-SQL style database which is not subject to this flaw may be chosen.
- Implementation: Use vigorous white-list style checking on any user input that may be used in an SQL command. Rather than escape meta-characters, it is safest to disallow them entirely. Reason: Later use of data that has been entered in the database may neglect to escape meta-characters before use.

Discussion

SQL injection has become a common issue with database-driven web sites. The flaw is easily detected, and easily exploited, and as such, any site or software package with even a minimal user base is likely to be subject to an attempted attack of this kind.

Essentially, the attack is accomplished by placing a meta character into data input to then place SQL commands in the control plane, which did not exist there before. This flaw depends on the fact that SQL makes no real distinction between the control and data planes.

Examples

In SQL:

```
select id, firstname, lastname from writers
```

If one provided:

```
Firstname: evil’ex
Lastname: Newman
```

the query string becomes:

```
select id, firstname, lastname from authors where forename = ‘evil’ex’ and surname
=’newman’
```

which the database attempts to run as

```
Incorrect syntax near al’ as the database tried to execute evil.
```

The above SQL statement could be Coded in Java as:

```
String firstName = requests.getParameters(“firstName”);
String lasttName = requests.getParameters(“firstName”);
PreparedStatement writersAdd = conn.prepareStatement(“SELECT id FROM writers WHERE
firstname=firstName”);
```

In which some of the same problems exist.

Related problems

- Injection problems

Deserialization of untrusted data

Overview

Data which is untrusted cannot be trusted to be well formed.

Consequences

- Availability: If a function is making an assumption on when to terminate, based on a sentry in a string, it could easily never terminate.
- Authorization: Potentially code could make assumptions that information in the deserialized object about the data is valid. Functions which make this dangerous assumption could be exploited.

Exposure period

- Requirements specification: A deserialization library could be used which provides a cryptographic framework to seal serialized data.
- Implementation: Not using the safe deserialization/serializing data features of a language can create data integrity problems.
- Implementation: Not using the protection accessor functions of an object can cause data integrity problems
- Implementation: Not protecting your objects from default overloaded functions — which may provide for raw output streams of objects — may cause data confidentiality problems.
- Implementation: Not making fields transient can often may cause data confidentiality problems.

Platform

- Languages: C, C++, Java
- Operating platforms: Any

Required resources

Any

Severity

Medium

Likelihood of exploit

Medium

Avoidance and mitigation

- Requirements specification: A deserialization library could be used which provides a cryptographic framework to seal serialized data.
- Implementation: Use the signing features of a language to assure that deserialized data has not been tainted.
- Implementation: When deserializing data populate a new object rather than just deserializing, the result is that the data flows through safe input validation and that the functions are safe.

- Implementation: Explicitly define final readObject() to prevent deserialization.

An example of this is:

```
private final void readObject(ObjectInputStream in)
throws java.io.IOException {
     throw new java.io.IOException("Cannot be deserialized");
}
```

- *Implementation*: Make fields transient to protect them from deserialization.

Discussion

It is often convenient to serialize objects for convenient communication or to save them for later use. However, deserialized data or code can often be modified without using the provided accessor functions if it does not use cryptography to protect itself. Furthermore, any cryptography would still be client-side security — which is of course a dangerous security assumption.

An attempt to serialize and then deserialize a class containing transient fields will result in NULLs where the non-transient data should be. This is an excellent way to prevent time, environment-based, or sensitive variables from being carried over and used improperly.

Examples

In Java:

```
try {
  File file = new File("object.obj");
  ObjectInputStream in = new ObjectInputStream(new
      FileInputStream(file));
  javax.swing.JButton button = (javax.swing.JButton)
      in.readObject();
  in.close();
  byte[] bytes = getBytesFromFile(file);
  in = new ObjectInputStream(new ByteArrayInputStream(bytes));
  button = (javax.swing.JButton) in.readObject();
  in.close();
}
```

Related problems

Not available.

Category 2: Environmental Problems

This section introduces the vulnerability Problem Types organized under the problem type “environmental problems.”

Reliance on data layout

Overview

Assumptions about protocol data or data stored in memory can be invalid, resulting in using data in ways that were unintended.

Consequences

Access control (including confidentiality and integrity): Can result in unintended modifications or information leaks of data.

Exposure period

Design: This problem can arise when a protocol leaves room for interpretation and is implemented by multiple parties that need to interoperate.

Implementation: This problem can arise by not understanding the subtleties either of writing portable code or of changes between protocol versions.

Platform

Protocol errors of this nature can happen on any platform. Invalid memory layout assumptions are possible in languages and environments with a single, flat memory space, such as C/C++ and Assembly.

Required resources

Any

Severity

Medium to High

Likelihood of exploit

Low

Avoidance and mitigation

- Design and Implementation: In flat address space situations, never allow computing memory addresses as offsets from another memory address.
- Design: Fully specify protocol layout unambiguously, providing a structured grammar (e.g., a compilable yacc grammar).
- Testing: Test that the implementation properly handles each case in the protocol grammar.

Discussion

When changing platforms or protocol versions, data may move in unintended ways. For example, some architectures may place local variables *a* and *b* right next to each other with *a* on top; some may place them next to each other with *b* on top; and others may add some padding to each. This ensured that each variable is aligned to a proper word size.

In protocol implementations, it is common to offset relative to another field to pick out a specific piece of data. Exceptional conditions — often involving new protocol versions — may add corner cases that lead to the data layout changing in an unusual way. The result can be that an implementation accesses a particular part of a packet, treating data of one type as data of another type.

Examples

In C:

```
void example() {
  char a;
  char b;
  *(&a + 1) = 0;
}
```

Here, *b* may not be one byte past *a*. It may be one byte in front of a. Or, they may have three bytes between them because they get aligned to 32-bit boundaries.

Related problems

Not available.

Relative path library search

Overview

Certain functions perform automatic path searching. The method and results of this path searching may not be as expected. Example: WinExec will use the space character as a delimiter, finding “C:\Program.exe” as an acceptable result for a search for “C:\Program Files\Foo\Bar.exe”.

Consequences

- Authorization: There is the potential for arbitrary code execution with privileges of the vulnerable program.

Exposure period

- Implementation: This flaw is a simple logic issue, introduced entirely at implementation time.

Platform

- Languages: Any
- Operating platforms*:* Any

Required resources

Any

Severity

High

Likelihood of exploit

High

Avoidance and mitigation

- Implementation: Use other functions which require explicit paths. Making use of any of the other readily available functions which require explicit paths is a safe way to avoid this problem.

Discussion

If a malicious individual has access to the file system, it is possible to elevate privileges by inserting such a file as “C:\Program.exe” to be run by a privileged program making use of WinExec.

Examples

In C\C++:

```
UINT errCode = WinExec(
  "C:\\Program Files\\Foo\\Bar",
  SW_SHOW
);
```

Related problems

Not available.

Relying on package-level scope

Overview

Java packages are not inherently closed; therefore, relying on them for code security is not a good practice.

Consequences

- Confidentiality: Any data in a Java package can be accessed outside of the Java framework if the package is distributed.
- Integrity: The data in a Java class can be modified by anyone outside of the Java framework if the packages is distributed.

Exposure period

Design through Implementation: This flaw is a style issue, so it is important to not allow direct access to variables and to protect objects.

Platform

- Languages: Java
- Operating platforms: Any

Required resources

Any

Severity

Medium

Likelihood of exploit

Medium

Avoidance and mitigation

- Design through Implementation: Data should be private static and final whenever possible. This will assure that your code is protected by instantiating early, preventing access and tampering.

Discussion

The purpose of package scope is to prevent accidental access. However, this protection provides an ease-of-software-development feature but not a security feature, unless it is sealed.

Examples

In Java:

```
package math;

public class Lebesgue implements Integration{

 public final Static String youAreHidingThisFunction(functionToIntegrate){
```

```
                return ...;
        }
```

Related problems

Not available.

Insufficient entropy in PRNG

Overview

The lack of entropy available for, or used by, a PRNG can be a stability and security threat.

Consequences

- Availability: If a pseudo-random number generator is using a limited entropy source which runs out (if the generator fails closed), the program may pause or crash.
- Authentication: If a PRNG is using a limited entropy source which runs out, and the generator fails open, the generator could produce predictable random numbers. Potentially a weak source of random numbers could weaken the encryption method used for authentication of users. In this case, potentially a password could be discovered.

Exposure period

- Design through Implementation: It is important — if one is utilizing randomness for important security — to use the best random numbers available.

Platform

- Languages: Any
- Operating platforms: Any

Required resources

Any

Severity

Medium

Likelihood of exploit

Medium

Avoidance and mitigation

- Implementation: Perform FIPS 140-1 tests on data to catch obvious entropy problems.
- Implementation: Consider a PRNG which re-seeds itself, as needed from a high quality pseudo-random output, like hardware devices.

Discussion

When deciding which PRNG to use, look at its sources of entropy. Depending on what your security needs are, you may need to use a random number generator which always uses strong random data — i.e., a random number generator which attempts to be strong but will fail in a weak way or will always provide some middle ground of protection through techniques like re-seeding. Generally something which always provides a predictable amount of strength is preferable and should be used.

Examples

In C/C++ or Java:

```
while (1){
  if (OnConnection()){
    if (PRNG(...)){
      //use the random bytes
    }
    else (PRNG(...)) {
      //cancel the program
  }
```

Related problems

Not available.

Failure of TRNG

Overview

True random number generators generally have a limited source of entropy and therefore can fail or block.

Consequences

- Availability: A program may crash or block if it runs out of random numbers.

Exposure period

- Requirements specification: Choose an operating system which is aggressive and effective at generating true random numbers.
- Implementation: This type of failure is a logical flaw which can be exacerbated by a lack of or the misuse of mitigating technologies.

Platform

- Languages: Any
- Operating platforms: Any

Required resources

Any

Severity

Medium

Likelihood of exploit

Low to Medium

Avoidance and mitigation

- Implementation: Rather than failing on a lack of random numbers, it is often preferable to wait for more numbers to be created.

Discussion

The rate at which true random numbers can be generated is limited. It is important that one uses them only when they are needed for security.

Examples

In C:

```
while (1){
  if (connection){
    if (hwRandom()){
      //use the random bytes
    }
    else (hwRandom()) {
```

```
                //cancel the program
            }
          }
```

Related problems

Not available.

Publicizing of private data when using inner classes

Overview

Java byte code has no notion of an inner class; therefore inner classes provide only a package-level security mechanism. Furthermore, the inner class gets access to the fields of its outer class even if that class is declared private.

Consequences

- Confidentiality: “Inner Classes” data confidentiality aspects can often be overcome.

Exposure period

Implementation: This is a simple logical flaw created at implementation time.

Platform

- Languages: Java
- Operating platforms: Any

Required resources

Any

Severity

Medium

Likelihood   of exploit

Medium

Avoidance and mitigation

- Implementation: Using sealed classes protects object-oriented encapsulation paradigms and therefore protects code from being extended in unforeseen ways.
- Implementation: Inner Classes do not provide security. Warning: Never reduce the security of the object from an outer class, going to an inner class. If your outer class is final or private, ensure that your inner class is private as well.

Discussion

A common misconception by Java programmers is that inner classes can only be accessed by outer classes. Inner classes’ main function is to reduce the size and complexity of code. This can be trivially broken by injecting byte code into the package. Furthermore, since an inner class has access to all fields in the outer class — even if the outer class is private — potentially access to the outer classes fields could be accidently compromised.

Examples

In Java:

```
private class Secure(){
        private password="mypassword"
        public class Insecure(){...}
    }
```

Related problems

Not available.

Trust of system event data

Overview

Security based on event locations are insecure and can be spoofed.

Consequences

- Authorization: If one trusts the system-event information and executes commands based on it, one could potentially take actions based on a spoofed identity.

Exposure period

- Design through Implementation: Trusting unauthenticated information for authentication is a design flaw.

Platform

- Languages: Any
- Operating platforms: Any

Required resources

Any

Severity

High

Likelihood of exploit

High

Avoidance and mitigation

- Design through Implementation: Never trust or rely any of the information in an Event for security.

Discussion

Events are a messaging system which may provide control data to programs listening for events. Events often do not have any type of authentication framework to allow them to be verified from a trusted source.

Any application, in Windows, on a given desktop can send a message to any window on the same desktop. There is no authentication framework for these messages. Therefore, any message can be used to manipulate any process on the desktop if the process does not check the validity and safeness of those messages.

Examples

In Java:

```
public void actionPerformed(ActionEvent e) {
  if (e.getSource()==button)
    System.out.println(“print out secret information”);
}
```

Related problems

Not available.

Resource exhaustion (file descriptor, disk space, sockets, ...)

Overview

Resource exhaustion is a simple denial of service condition which occurs when the resources necessary to perform an action are entirely consumed, therefore preventing that action from taking place.

Consequences

- Availability: The most common result of resource exhaustion is denial-of-service.
- Access control: In some cases it may be possible to force a system to “fail open” in the event of resource exhaustion.

Exposure period

- Design: Issues in system architecture and protocol design may make systems more subject to resource-exhaustion attacks.
- Implementation: Lack of low level consideration often contributes to the problem.

Platform

- Languages: All
- Platforms: All

Required resources

Any

Severity

Low to medium

Likelihood of exploit

Very high

Avoidance and mitigation

- Design: Design throttling mechanisms into the system architecture.
- Design: Ensure that protocols have specific limits of scale placed on them.
- Implementation: Ensure that all failures in resource allocation place the system into a safe posture.
- Implementation: Fail safely when a resource exhaustion occurs.

Discussion

Resource exhaustion issues are generally understood but are far more difficult to successfully prevent. Resources can be exploited simply by ensuring that the target machine must do much more work and consume more resources in order to service a request than the attacker must do to initiate a request.

Prevention of these attacks requires either that the target system:

- either recognizes the attack and denies that user further access for a given amount of time;

- or uniformly throttles all requests in order to make it more difficult to consume resources more quickly than they can again be freed.

The first of these solutions is an issue in itself though, since it may allow attackers to prevent the use of the system by a particular valid user. If the attacker impersonates the valid user, he may be able to prevent the user from accessing the server in question.

The second solution is simply difficult to effectively institute — and even when properly done, it does not provide a full solution. It simply makes the attack require more resources on the part of the attacker.

The final concern that must be discussed about issues of resource exhaustion is that of systems which “fail open.” This means that in the event of resource consumption, the system fails in such a way that the state of the system — and possibly the security functionality of the system — is compromised. A prime example of this can be found in old switches that were vulnerable to “macof” attacks (so named for a tool developed by Dugsong). These attacks flooded a switch with random IP and MAC address combinations, therefore exhausting the switch’s cache, which held the information of which port corresponded to which MAC addresses. Once this cache was exhausted, the switch would fail in an insecure way and would begin to act simply as a hub, broadcasting all traffic on all ports and allowing for basic sniffing attacks.

Examples

In Java:

```
class Worker implements Executor {
 ...
  public void execute(Runnable r) {
  try {
   ...
  }
  catch (InterruptedException ie) { // postpone response
   Thread.currentThread().interrupt();
  }
 }

 public Worker(Channel ch, int nworkers) {
  ...
  }

 protected void activate() {
  Runnable loop = new Runnable() {
   public void run() {
    try {
     for (;;) {
      Runnable r = ...
      r.run();
     }
    }
    catch (InterruptedException ie) {...}
   }
  };
  new Thread(loop).start();
 }
In C/C++:

int main(int argc, char *argv[]) {
  sock=socket(AF_INET, SOCK_STREAM, 0);
  while (1) {
    newsock=accept(sock, ...);
    printf("A connection has been accepted\n");
    pid = fork();
  }
```

There are no limits to runnables/forks. Potentially an attacker could cause resource problems very quickly.

Related problems

Not available.

Information leak through class cloning

Overview

Cloneable classes are effectively open classes since data cannot be hidden in them.

Consequences

- Confidentiality: A class which can be cloned can be produced without executing the constructor.

Exposure period

- Implementation: This is a style issue which needs to be adopted throughout the implementation of each class.

Platform

- Languages: Java
- Operating platforms: Any

Required resources

Any

Severity

Medium

Likelihood of exploit

Medium

Avoidance and mitigation

- Implementation: Make classes uncloneable by defining a clone function like:

```
public final void clone() throws java.lang.CloneNotSupportedException {
     throw new java.lang.CloneNotSupportedException();
}
```

- Implementation: If you do make your classes cloneable, ensure that your clone method is final and throw super.clone().

Discussion

Classes which do no explicitly deny cloning can be cloned by any other class without running the constructor. This is, of course, dangerous since numerous checks and security aspects of an object are often taken care of in the constructor.

Examples

```
public class CloneClient
{
        public CloneClient()
//throws java.lang.CloneNotSupportedException
        {
                Teacher t1 = new Teacher("guddu","22,nagar road");
    //...// Due some stuff to remove the teacher.
                Teacher t2 = (Teacher)t1.clone();
                                System.out.println(t2.name);
       }
        public static void main(String args[])
        {
                new CloneClient();
        }
}

class Teacher implements Cloneable
{
        public Object clone() {
                try { return super.clone();
                } catch (java.lang.CloneNotSupportedException e) {
                    throw new RuntimeException(e.toString());
                }
        }
        public String name;
        public String clas;
        public Teacher(String name,String clas)
        {
                this.name = name;
                this.clas = clas;

        }
}
```

Related problems

Not available.

Information leak through serialization

Overview

Serializable classes are effectively open classes since data cannot be hidden in them.

Consequences

- Confidentiality: Attacker can write out the class to a byte stream in which they can extract the important data from it.

Exposure period

- Implementation: This is a style issue which needs to be adopted throughout the implementation of each class.

Platform

- Languages: Java, C++
- Operating platforms: Any

Required resources

Any

Severity

High

Likelihood of exploit

High

Avoidance and mitigation

- Implementation: In Java, explicitly define final writeObject() to prevent serialization. This is the recommended solution. Define the writeObject() function to throw an exception explicitly denying serialization.
- Implementation: Make sure to prevent serialization of your objects.

Discussion

Classes which do no explicitly deny serialization can be serialized by any other class which can then in turn use the data stored inside it.

Examples

```
class Teacher
{

        private String name;
        private String clas;
        public Teacher(String name,String clas)
        {
               //...//Check the database for the name and address
                this.SetName() = name;
```

```
                        this.Setclas() = clas;

                }
        }
```

Related problems

Not available.

Overflow of static internal buffer

Overview

A non-final static field can be viewed and edited in dangerous ways.

Consequences

- Integrity: The object could potentially be tampered with.
- Confidentiality: The object could potentially allow the object to be read.

Exposure period

- Design through Implementation: This is a simple logical issue which can be easily remedied through simple protections.

Platform

- Languages: Java, C++
- Operating platforms: Any

Required resources

Any

Severity

Medium

Likelihood   of exploit

High

Avoidance and mitigation

- Design through Implementation: Make any static fields private and final.

Discussion

Non-final fields, which are not public can be read and written to by arbitrary Java code.

Examples

In C++:

```
public int password r = 45;
```

In Java:

```
static public String r;
```

This is a uninitiated static class which can be accessed without a get-accessor and changed without a set-accessor.

Related problems

Not available.

Category 3: Synchronization & Timing Errors

This section introduces the vulnerability Problem Types organized under the problem type "synchronization and timing errors."

State synchronization error

Overview

State synchronization refers to a set of flaws involving contradictory states of execution in a process which result in undefined behavior.

Consequences

- Undefined: Depending on the nature of the state of corruption, any of the listed consequences may result.

Exposure period

- Design: Design flaws may be to blame for out-of-sync states, but this is the rarest method.
- Implementation: Most likely, state-synchronization errors occur due to logical flaws and race conditions introduced at implementation time.
- Run time: Hardware, operating system, or interaction with other programs may lead to this error.

Platform

- Languages: All
- Operating platforms: All

Required resources

Any

Severity

High

Likelihood of exploit

Medium to High

Avoidance and mitigation

- Implementation: Pay attention to asynchronous actions in processes; and make copious use of sanity checks in systems that may be subject to synchronization errors.

Discussion

The class of synchronization errors is large and varied, but all rely on the same essential flaw. The state of the system is not what the process expects it to be at a given time.

Obviously, the range of possible symptoms is enormous, as is the range of possible solutions. The flaws presented in this section are some of the most difficult to diagnose and fix. It is more important to know how to characterize specific flaws than to gain information about them.

Examples

In C/C++:

```
static void print(char * string) {
  char * word;
  int counter;
  fflush(stdout);
  for(word = string; counter = *word++; ) putc(counter, stdout);
}

int main(void) {
   pid_t pid;
   if( (pid = fork()) < 0) exit(-2);
   else if( pid == 0) print("child");
   else print("parent\n");
   exit(0);
}
```

In Java:

```
class read{
  private int lcount;
  private int rcount;
  private int wcount;

  public void getRead(){
    while ((lcount == -1) || (wcount !=0));
    lcount++;

  public void getWrite(){
    while ((lcount == -0);
    lcount--;
    lcount=-1;

  public void killLocks(){
    if (lcount==0) return;
    else if (lcount == -1) lcount++;
    else lcount--;
  }
}
```

Related problems

Not available.

Covert timing channel

Overview

Unintended information about data gets leaked through observing the timing of events.

Consequences

- Confidentiality: Information leakage.

Exposure period

- Design: Protocols usually have timing difficulties implicit in their design.
- Implementation: Sometimes a timing covert channel can be dependent on implementation strategy. Example: Using conditionals may leak information, but using table lookup will not.

Platform

Any

Required resources

Any

Severity

Medium

Likelihood   of exploit

Medium

Avoidance and mitigation

- Design: Whenever possible, specify implementation strategies that do not introduce time variances in operations.
- Implementation: Often one can artificially manipulate the time which operations take or — when operations occur — can remove information from the attacker.

Discussion

Sometimes simply knowing when data is sent between parties can provide a malicious user with information that should be unauthorized.

Other times, externally monitoring the timing of operations can reveal sensitive data. For example, some cryptographic operations can leak their internal state if the time it takes to perform the operation changes, based on the state. In such cases, it is good to switch algorithms or implementation techniques. It is also reasonable to add artificial stalls to make the operation take the same amount of raw CPU time in all cases.

Examples

In Python:

```
def validate_password(actual_pw, typed_pw):
```

```
if len(actual_pw) <> len(typed_pw):
  return 0
for i in len(actual_pw):
  if actual_pw[i] <> typed_pw[i]:
  return 0
return 1
```

In this example, the attacker can observe how long an authentication takes when the user types in the correct password. When the attacker tries his own values, he can first try strings of various length. When he finds a string of the right length, the computation will take a bit longer because the *for* loop will run at least once.

Additionally, with this code, the attacker can possibly learn one character of the password at a time, because when he guesses the first character right, the computation will take longer than when he guesses wrong. Such an attack can break even the most sophisticated password with a few hundred guesses.

Note that, in this example, the actual password must be handled in constant time, as far as the attacker is concerned, even if the actual password is of an unusual length. This is one reason why it is good to use an algorithm that, among other things, stores a seeded cryptographic one-way hash of the password, then compare the hashes, which will always be of the same length.

Related problems

- Storage covert channel

Symbolic name not mapping to correct object

Overview

A constant symbolic reference to an object is used, even though the underlying object changes over time.

Consequences

- Access control: The attacker can gain access to otherwise unauthorized resources.
- Authorization: Race conditions such as this kind may be employed to gain read or write access to resources not normally readable or writable by the user in question.
- Integrity: The resource in question, or other resources (through the corrupted one) may be changed in undesirable ways by a malicious user.
- Accountability: If a file or other resource is written in this method, as opposed to a valid way, logging of the activity may not occur.
- Non-repudiation: In some cases it may be possible to delete files that a malicious user might not otherwise have access to — such as log files.

Exposure period

Platform

Required resources

Severity

Likelihood   of exploit

Avoidance and mitigation

Discussion

See more specific instances.

Examples

Not available.

Related problems

- Time of check, time of use race condition
- Comparing classes by name

Time of check, time of use race condition

Overview

Time-of-check, time-of-use race conditions occur when between the time in which a given resource is checked, and the time that resource is used, a change occurs in the resource to invalidate the results of the check.

Consequences

- Access control: The attacker can gain access to otherwise unauthorized resources.
- Authorization: race conditions such as this kind may be employed to gain read or write access to resources which are not normally readable or writable by the user in question.
- Integrity: The resource in question, or other resources (through the corrupted one), may be changed in undesirable ways by a malicious user.
- Accountability: If a file or other resource is written in this method, as opposed to in a valid way, logging of the activity may not occur.
- Non-repudiation: In some cases it may be possible to delete files a malicious user might not otherwise have access to, such as log files.

Exposure period

- Design: Strong locking methods may be designed to protect against this flaw.
- Implementation: Use of system APIs may prevent check, use race conditions.

Platform

- Languages: Any
- Platforms: All

Required resources

- Some access to the resource in question

Severity

Medium

Likelihood of exploit

Low to Medium

Avoidance and mitigation

- Design: Ensure that some environmental locking mechanism can be used to protect resources effectively.
- Implementation: Ensure that locking occurs before the check, as opposed to afterwards, such that the resource, as checked, is the same as it is when in use.

Discussion

Time-of-check, time-of-use race conditions occur when a resource is checked for a particular value, that value is changed, then the resource is used, based on the assumption that the value is still the same as it was at check time.

This is a broad category of race condition encompassing binding flaws, locking race conditions, and others.

Examples

In C/C++:

```
struct stat *sb;
..
lstat(“...”,sb);
// it has not been updated since the last time it was read
printf(“stated file\n”);
if (sb->st_mtimespec==...)
  print(“Now updating things\n”);
  updateThings();
}
```

Potentially the file could have been updated between the time of the check and the lstat, especially since the printf has latency.

Related problems

- State synchronization error

Comparing classes by name

Overview

The practice of determining an object's type, based on its name, is dangerous since malicious code may purposely reuse class names in order to appear trusted.

Consequences

- Authorization: If a program trusts, based on the name of the object, to assume that it is the correct object, it may execute the wrong program.

Exposure period

- Implementation: This flaw is a simple logic issue, introduced entirely at implementation time.

Platform

- Languages: Java
- Operating platforms: Any

Required resources

Any

Severity

High

Likelihood of exploit

High

Avoidance and mitigation

- Implementation: Use class equivalency to determine type. Rather than use the class name to determine if an object is of a given type, use the getClass() method, and == operator.

Discussion

If the decision to trust the methods and data of an object is based on the name of a class, it is possible for malicious users to send objects of the same name as trusted classes and thereby gain the trust afforded to known classes and types.

Examples

```
if (inputClass.getClass().getName().equals(“TrustedClassName”)) {
  // Do something assuming you trust inputClass
  // …
}
```

Related problems

Not available.

Race condition in switch

Overview

If the variable which is switched on is changed while the switch statement is still in progress, undefined activity may occur.

Consequences

- Undefined: This flaw will result in the system state going out of sync.

Exposure period

- Implementation: Variable locking is the purview of implementers.

Platform

- Languages: All that allow for multi-threaded activity
- Operating platforms: All

Required resources

Any

Severity

Medium

Likelihood of exploit

Medium

Avoidance and mitigation

- Implementation: Variables that may be subject to race conditions should be locked for the duration of any switch statements.

Discussion

This issue is particularly important in the case of switch statements that involve fall-through style case statements — i.e., those which do not end with break.

If the variable which we are switching on change in the course of execution, the actions carried out may place the state of the process in a contradictory state or even result in memory corruption.

For this reason, it is important to ensure that all variables involved in switch statements are locked before the statement starts and are unlocked when the statement ends.

Examples

In C/C++:

```
#include <sys/types.h>
#include <sys/stat.h>

int main(argc,argv){
```

```
                struct stat *sb;
                time_t timer;

                lstat("bar.sh",sb);

                printf("%d\n",sb->st_ctime);
                switch(sb->st_ctime % 2){
                        case 0: printf("One option\n");break;
                        case 1: printf("another option\n");break;
                        default: printf("huh\n");break;
                }

                return 0;
        }
```

Related problems

- Race condition in signal handler
- Race condition within a thread

Race condition in signal handler

Overview

Race conditions occur frequently in signal handlers, since they are asynchronous actions. These race conditions may have any number of Problem Types and symptoms.

Consequences

- Authorization: It may be possible to execute arbitrary code through the use of a write-what-where condition.
- Integrity: Signal race conditions often result in data corruption.

Exposure period

- Requirements specification: A language might be chosen which is not subject to this flaw.
- Design: Signal handlers with complicated functionality may result in this issue.
- Implementation: The use of any non-reentrant functionality or global variables in a signal handler might result in this race conditions.

Platform

- Languages: C, C++, Assembly
- Operating platforms: All

Required resources

Any

Severity

High

Likelihood of exploit

Medium

Avoidance and mitigation

- Requirements specification: A language might be chosen, which is not subject to this flaw, through a guarantee of reentrant code.
- Design: Design signal handlers to only set flags rather than perform complex functionality.
- Implementation: Ensure that non-reentrant functions are not found in signal handlers. Also, use sanity checks to ensure that state is consistent be performing asynchronous actions which effect the state of execution.

Discussion

Signal race conditions are a common issue that have only recently been seen as exploitable. These issues occur when non-reentrant functions, or state-sensitive actions occur in the signal handler, where they may be called at any time. If these functions are called at an inopportune moment — such as while a non-reentrant function is already running —, memory corruption occurs that may be exploitable.

Another signal race condition commonly found occurs when free is called within a signal handler, resulting in a double free and therefore a write-what-where condition. This is a perfect example of a signal handler taking actions which cannot be accounted for in state. Even if a given pointer is set to NULL after it has been freed, a race condition still exists between the time the memory was freed and the pointer was set to NULL. This is especially prudent if the same signal handler has been set for more than one signal — since it means that the signal handler itself may be reentered.

Examples

```
#include <signal.h>
#include <syslog.h>
#include <string.h>
#include <stdlib.h>

void *global1, *global2;
char *what;

void sh(int dummy) {
  syslog(LOG_NOTICE,"%s\n",what);
  free(global2);
  free(global1);
  sleep(10);
  exit(0);
}

int main(int argc,char* argv[]) {
  what=argv[1];
  global1=strdup(argv[2]);
  global2=malloc(340);
  signal(SIGHUP,sh);
  signal(SIGTERM,sh);
  sleep(10);
  exit(0);
}
```

Related problems

- Doubly freeing memory
- Using freed memory
- Unsafe function call from a signal handler
- Write-what-where

Unsafe function call from a signal handler

Overview

There are several functions which — under certain circumstances, if used in a signal handler — may result in the corruption of memory, allowing for exploitation of the process.

Consequences

- Access control: It may be possible to execute arbitrary code through the use of a write-what-where condition.
- Integrity: Signal race conditions often result in data corruption.

Exposure period

- Requirements specification: A language might be chosen which is not subject to this flaw.
- Design: Signal handlers with complicated functionality may result in this issue.
- Implementation: The use of any number of non-reentrant functions will result in this issue.

Platform

- Languages: C, C++, Assembly
- Platforms: All

Required resources

Any

Severity

High

Likelihood of exploit

Low

Avoidance and mitigation

- Requirements specification: A language might be chosen, which is not subject to this flaw, through a guarantee of reentrant code.
- Design: Design signal handlers to only set flags rather than perform complex functionality.
- Implementation: Ensure that non-reentrant functions are not found in signal handlers. Also, use sanity checks to ensure that state is consistently performing asynchronous actions which effect the state of execution.

Discussion

This flaw is a subset of race conditions occurring in signal handler calls which is concerned primarily with memory corruption caused by calls to non-reentrant functions in signal handlers.

Non-reentrant functions are functions that cannot safely be called, interrupted, and then recalled before the first call has finished without resulting in memory corruption. The function call syslog() is an example of this. In order to perform its functionality, it allocates a small amount of memory as “scratch space.” If syslog() is suspended by a signal call and the signal handler calls syslog(), the memory used by both of these functions enters an undefined, and possibly, exploitable state.

Examples

See *Race condition in signal handler*, for an example usage of free() in a signal handler which is exploit-able.

Related problems

- Race condition in signal handler
- Write-what-where

Failure to drop privileges when reasonable

Overview

Failing to drop privileges when it is reasonable to do so results in a lengthened time during which exploitation may result in unnecessarily negative consequences.

Consequences

- Access control: An attacker may be able to access resources with the elevated privilege that he should not have been able to access. This is particularly likely in conjunction with another flaw — e.g., a buffer overflow.

Exposure period

- Design: Privilege separation decisions should be made and enforced at the architectural design phase of development.

Platform

- Languages: Any
- Platforms: All

Required resources

Any

Severity

High

Likelihood of exploit

Undefined.

Avoidance and mitigation

- Design: Ensure that appropriate compartmentalization is built into the system design and that the compartmentalization serves to allow for and further reinforce privilege separation functionality. Architects and designers should rely on the principle of least privilege to decide when it is appropriate to use and to drop system privileges.

Discussion

The failure to drop system privileges when it is reasonable to do so is not a vulnerability by itself. It does, however, serve to significantly increase the Severity of other vulnerabilities. According to the principle of least privilege, access should be allowed only when it is absolutely necessary to the function of a given system, and only for the minimal necessary amount of time.

Any further allowance of privilege widens the window of time during which a successful exploitation of the system will provide an attacker with that same privilege.

If at all possible, limit the allowance of system privilege to small, simple sections of code that may be called atomically.

Examples

In C/C++:

```
setuid(0);
//Do some important stuff
//setuid(old_uid);
// Do some non privlidged stuff.
```

In Java:

```
method() {
  AccessController.doPrivileged(new PrivilegedAction() {
      public Object run() {
      //Insert all code here
      }
        });
}
```

Related problems

- All problems with the consequence of “Access control.”

Race condition in checking for certificate revocation

Overview

If the revocation status of a certificate is not checked before each privilege requiring action, the system may be subject to a race condition, in which their certificate may be used before it is checked for revocation.

Consequences

- Authentication: Trust may be assigned to an entity who is not who it claims to be.
- Integrity: Data from an untrusted (and possibly malicious) source may be integrated.
- Confidentiality: Date may be disclosed to an entity impersonating a trusted entity, resulting in information disclosure.

Exposure period

- Design: Checks for certificate revocation should be included in the design of a system
- Design: One can choose to use a language which abstracts out this part of the authentication process.

Platform

- Languages: Languages which do not abstract out this part of the process.
- Operating platforms: All

Required resources

Minor trust: Users must attempt to interact with the malicious system.

Severity

Medium

Likelihood of exploit

Medium

Avoidance and mitigation

- Design: Ensure that certificates are checked for revoked status before each use of a protected resource

Discussion

If a certificate is revoked after the initial check, all subsequent actions taken with the owner of the revoked certificate will loose all benefits guaranteed by the certificate. In fact, it is almost certain that the use of a revoked certificate indicates malicious activity.

If the certificate is checked before each access of a protected resource, the delay subject to a possible race condition becomes almost negligible and significantly reduces the risk associated with this issue.

Examples

In C/C++:

```
if (!(cert = SSL_get_peer(certificate(ssl)) || !host)
  foo=SSL_get_veryify_result(ssl);
  if (X509_V_OK==foo)
//do stuff
  foo=SSL_get_veryify_result(ssl);
 //do more stuff without the check.
```

Related problems

- Failure to follow chain of trust in certificate validation
- Failure to validate host-specific certificate data
- Failure to validate certificate expiration
- Failure to check for certificate revocation

Passing mutable objects to an untrusted method

Overview

Sending non-cloned mutable data as an argument may result in that data being altered or deleted by the called function, thereby putting the calling function into an undefined state.

Consequences

- Integrity: Potentially data could be tampered with by another function which should not have been tampered with.

Exposure period

- Implementation: This flaw is a simple logic issue, introduced entirely at implementation time.

Platform

- Languages: C/C++ or Java
- Operating platforms: Any

Required resources

Any

Severity

Medium

Likelihood   of exploit

Medium

Avoidance and mitigation

- Implementation: Pass in data which should not be alerted as constant or immutable.
- Implementation: Clone all mutable data before returning references to it. This is the preferred mitigation. This way — regardless of what changes are made to the data — a valid copy is retained for use by the class.

Discussion

In situations where unknown code is called with references to mutable data, this external code may possibly make changes to the data sent. If this data was not previously cloned, you will be left with modified data which may, or may not, be valid in the context of execution.

Examples

In C\C++:

```
private:
  int foo.
  complexType bar;
  String baz;
  otherClass externalClass;
```

```
          public:
            void doStuff() {
              externalClass.doOtherStuff(foo, bar, baz)
            }
```

In this example, *bar* and *baz* will be passed by reference to doOtherStuff() which may change them.

Related problems

Not available.

Mutable object returned

Overview

Sending non-cloned mutable data as a return value may result in that data being altered or deleted by the called function, thereby putting the class in an undefined state.

Consequences

- Access Control / Integrity: Potentially data could be tampered with by another function which should not have been tampered with.

Exposure period

- Implementation: This flaw is a simple logic issue, introduced entirely at implementation time.

Platform

- Languages: C,C++ or Java
- Operating platforms: Any

Required resources

Any

Severity

Medium

Likelihood   of exploit

Medium

Avoidance and mitigation

- Implementation: Pass in data which should not be alerted as constant or immutable.
- Implementation: Clone all mutable data before returning references to it. This is the preferred mitigation. This way, regardless of what changes are made to the data, a valid copy is retained for use by the class.

Discussion

In situations where functions return references to mutable data, it is possible that this external code, which called the function, may make changes to the data sent. If this data was not previously cloned, you will be left with modified data which may, or may not, be valid in the context of the class in question.

Examples

In C\C++:

```
private:
  externalClass foo;

public:
  void doStuff() {
```

```
//..//Modify foo
    return foo;
  }
```

In Java:

```
public class foo {
 private externalClass bar = new externalClass();
 public doStuff(...){
   //..//Modify bar
   return bar;
 }
```

Related problems

Not available.

Accidental leaking of sensitive information through error messages

Overview

Server messages need to be parsed before being passed on to the user.

Consequences

- Confidentiality: Often this will either reveal sensitive information which may be used for a later attack or private information stored in the server.

Exposure period

- Implementation: This flaw is a simple logic issue, introduced entirely at implementation time.
- Build: It is important to adequately set read privileges and otherwise operationally protect the log.

Platform

- Languages: Any; it is especially prevalent, however, when dealing with SQL or languages which throw errors.
- Operating platforms: Any

Required resources

Any

Severity

High

Likelihood   of exploit

High

Avoidance and mitigation

- Implementation: Any error should be parsed for dangerous revelations.
- Build: Debugging information should not make its way into a production release.

Discussion

The first thing an attacker may use — once an attack has failed — to stage the next attack is the error information provided by the server.

SQL Injection attacks generally probe the server for information in order to stage a successful attack.

Examples

In Java:

```
try {
  /.../
} catch (Exception e) {
```

```
            System.out.println(e);
          }
```

Here you are passing much more data than is needed.

Another example is passing the SQL exceptions to a WebUser without filtering.

Related problems

Not available.

Accidental leaking of sensitive information through sent data

Overview

The accidental leaking of sensitive information through sent data refers to the transmission of data which is either sensitive in and of itself or useful in the further exploitation of the system through standard data channels.

Consequences

- Confidentiality: Data leakage results in the compromise of data confidentiality.

Exposure period

- Requirements specification: Information output may be specified in the requirements documentation.
- Implementation: The final decision as to what data is sent is made at implementation time.

Platform

- Languages: All
- Platforms: All

Required resources

Any

Severity

Low

Likelihood of exploit

Undefined.

Avoidance and mitigation

- Requirements specification: Specify data output such that no sensitive data is sent.
- Implementation: Ensure that any possibly sensitive data specified in the requirements is verified with designers to ensure that it is either a calculated risk or mitigated elsewhere.

Discussion

Accidental data leakage occurs in several places and can essentially be defined as unnecessary data leakage. Any information that is not necessary to the functionality should be removed in order to lower both the overhead and the possibility of security sensitive data being sent.

Examples

The following is an actual mysql error statement:

```
Warning: mysql_pconnect():
Access denied for user: 'root@localhost' (Using password: N1nj4) in /usr/local/www/wi-
data/includes/database.inc on line 4
```

Related problems

- Accidental leaking of sensitive information through error messages
- Accidental leaking of sensitive information through data queries

Accidental leaking of sensitive information through data queries

Overview

When trying to keep information confidential, an attacker can often infer some of the information by using statistics.

Consequences

- Confidentiality: Sensitive information may possibly be through data queries accidentally.

Exposure period

- Design: Proper mechanisms for preventing this kind of problem generally need to be identified at the design level.

Platform

Any; particularly systems using relational databases or object-relational databases.

Required resources

Any

Severity

Medium

Likelihood of exploit

Medium

Avoidance and mitigation

This is a complex topic. See the book *Translucent Databases* for a good discussion of best practices.

Discussion

In situations where data should not be tied to individual users, but a large number of users should be able to make queries that “scrub” the identity of users, it may be possible to get information about a user — e.g., by specifying search terms that are known to be unique to that user.

Examples

See the book *Translucent Databases* for examples.

Related problems

Not available.

Race condition within a thread

Overview

If two threads of execution use a resource simultaneously, there exists the possibility that resources may be used while invalid, in turn making the state of execution undefined.

Consequences

- Integrity: The main problem is that — if a lock is overcome — data could be altered in a bad state.

Exposure period

- Design: Use a language which provides facilities to easily use threads safely.

Platform

- Languages: Any language with threads
- Operating platforms: All

Required resources

Any

Severity

High

Likelihood of exploit

Medium

Avoidance and mitigation

Discussion

- Design: Use locking functionality. This is the recommended solution. Implement some form of locking mechanism around code which alters or reads persistent data in a multi-threaded environment.
- Design: Create resource-locking sanity checks. If no inherent locking mechanisms exist, use flags and signals to enforce your own blocking scheme when resources are being used by other threads of execution.

Examples

In C/C++:

```
int foo = 0;
    int storenum(int num)
    {
        static int counter = 0;
        counter++;
        if (num > foo)
            foo = num;
            return foo;
    }
```

In Java:

```
public classRace {
  static int foo = 0;

  public static void main() {
    new Threader().start();
    foo = 1;
  }

  public static class Threader extends Thread {
    public void run() {
      System.out.println(foo);
    }
  }
}
```

Related problems

Not available.

Reflection attack in an auth protocol

Overview

Simple authentication protocols are subject to reflection attacks if a malicious user can use the target machine to impersonate a trusted user.

Consequences

- Authentication: The primary result of reflection attacks is successful authentication with a target machine — as an impersonated user.

Exposure period

- Design: Protocol design may be employed more intelligently in order to remove the possibility of reflection attacks.

Platform

- Languages: Any
- Platforms: All

Required resources

Any

Severity

Medium to High

Likelihood of exploit

Medium

Avoidance and mitigation

- Design: Use different keys for the initiator and responder or of a different type of challenge for the initiator and responder.

Discussion

Reflection attacks capitalize on mutual authentication schemes in order to trick the target into revealing the secret shared between it and another valid user.

In a basic mutual-authentication scheme, a secret is known to both the valid user and the server; this allows them to authenticate. In order that they may verify this shared secret without sending it plainly over the wire, they utilize a Diffie-Hellman-style scheme in which they each pick a value, then request the hash of that value as keyed by the shared secret.

In a reflection attack, the attacker claims to be a valid user and requests the hash of a random value from the server. When the server returns this value and requests its own value to be hashed, the attacker opens another connection to the server. This time, the hash requested by the attacker is the value which the server requested in the first connection. When the server returns this hashed value, it is used in the first connection, authenticating the attacker successfully as the impersonated valid user.

Examples

In C/C++:

```
unsigned char *simple_digest(char *alg,char *buf,unsigned int len, int *olen) {
        const EVP_MD *m;
        EVP_MD_CTX ctx;
        unsigned char *ret;

        OpenSSL_add_all_digests();
        if (!(m = EVP_get_digestbyname(alg)))
                return NULL;
        if (!(ret = (unsigned char*)malloc(EVP_MAX_MD_SIZE)))
                return NULL;
        EVP_DigestInit(&ctx, m);
        EVP_DigestUpdate(&ctx,buf,len);
        EVP_DigestFinal(&ctx,ret,olen);
        return ret;
}

unsigned char *generate_password_and_cmd(char *password_and_cmd){
        simple_digest("sha1",password,strlen(password_and_cmd)...);
}
```

In Java:

```
String command = new String(“some cmd to execute & the password”)
MessageDigest encer = MessageDigest.getInstance("SHA");
encer.update(command.getBytes(“UTF-8”));
byte[] digest = encer.digest();
```

Related problems

- Using a broken or risky cryptographic algorithm

Capture-replay

Overview

A capture-relay protocol flaw exists when it is possible for a malicious user to sniff network traffic and replay it to the server in question to the same effect as the original message (or with minor changes).

Consequences

- Authorization: Messages sent with a capture-relay attack allow access to resources which are not otherwise accessible without proper authentication.

Exposure period

- Design: Prevention of capture-relay attacks must be performed at the time of protocol design.

Platform

- Languages: All
- Operating platforms: All

Required resources

Network proximity: Some ability to sniff from, and inject messages into, a stream would be required to capitalize on this flaw.

Severity

Medium to High

Likelihood of exploit

High

Avoidance and mitigation

- Design: Utilize some sequence or time stamping functionality along with a checksum which takes this into account in order to ensure that messages can be parsed only once.

Discussion

Capture-relay attacks are common and can be difficult to defeat without cryptography. They are a subset of network injection attacks that rely listening in on previously sent valid commands, then changing them slightly if necessary and resending the same commands to the server.

Since any attacker who can listen to traffic can see sequence numbers, it is necessary to sign messages with some kind of cryptography to ensure that sequence numbers are not simply doctored along with content.

Examples

In C/C++:

```
unsigned char *simple_digest(char *alg,char *buf,unsigned int len, int *olen) {
        const EVP_MD *m;
        EVP_MD_CTX ctx;
        unsigned char *ret;

        OpenSSL_add_all_digests();
        if (!(m = EVP_get_digestbyname(alg)))
                return NULL;
        if (!(ret = (unsigned char*)malloc(EVP_MAX_MD_SIZE)))
                return NULL;
        EVP_DigestInit(&ctx, m);
        EVP_DigestUpdate(&ctx,buf,len);
        EVP_DigestFinal(&ctx,ret,olen);
        return ret;
}

unsigned char *generate_password_and_cmd(char *password_and_cmd){
        simple_digest("sha1",password,strlen(password_and_cmd)...);
}
```

In Java:

```
String command = new String(“some cmd to execute & the password”)
MessageDigest encer = MessageDigest.getInstance("SHA");
encer.update(command.getBytes(“UTF-8”));
byte[] digest = encer.digest();
```

Related problems

Not available.

Category 4: Protocol Errors

This section introduces the vulnerability Problem Types organized under the problem type “protocol errors.”

Failure to follow chain of trust in certificate validation

Overview

Failure to follow the chain of trust when validating a certificate results in the trust of a given resource which has no connection to trusted root-certificate entities.

Consequences

- Authentication: Exploitation of this flaw can lead to the trust of data that may have originated with a spoofed source.
- Accountability: Data, requests, or actions taken by the attacking entity can be carried out as a spoofed benign entity.

Exposure period

- Design: Proper certificate checking should be included in the system design.
- Implementation: If use of SSL (or similar) is simply mandated by design and requirements, it is the implementor’s job to properly use the API and all its protections.

Platform

- Languages: All
- Platforms: All

Required resources

Minor trust: Users must attempt to interact with the malicious system.

Severity

Medium

Likelihood of exploit

Low

Avoidance and mitigation

- Design: Ensure that proper certificate checking is included in the system design.
- Implementation: Understand, and properly implement all checks necessary to ensure the integrity of certificate trust integrity.

Discussion

If a system fails to follow the chain of trust of a certificate to a root server, the certificate looses all usefulness as a metric of trust. Essentially, the trust gained from a certificate is derived from a chain of trust — with a reputable trusted entity at the end of that list. The end user must trust that reputable source, and this reputable source must vouch for the resource in question through the medium of the certificate.

In some cases, this trust traverses several entities who vouch for one another. The entity trusted by the end user is at one end of this trust chain, while the certificate wielding resource is at the other end of the chain.

If the user receives a certificate at the end of one of these trust chains and then proceeds to check only that the first link in the chain, no real trust has been derived, since you must traverse the chain to a trusted source to verify the certificate.

Examples

```
if (!(cert = SSL_get_peer(certificate(ssl)) || !host)
  foo=SSL_get_veryify_result(ssl);
  if ((X509_V_OK==foo) || X509_V_ERR_SELF_SIGNED_CERT_IN_CHAIN==foo))
//do stuff
```

Related problems

- Key exchange without entity authentication
- Failure to validate host-specific certificate data
- Failure to validate certificate expiration
- Failure to check for certificate revocation

Key exchange without entity authentication

Overview

Performing a key exchange without verifying the identity of the entity being communicated with will preserve the integrity of the information sent between the two entities; this will not, however, guarantee the identity of end entity.

Consequences

- Authentication: No authentication takes place in this process, bypassing an assumed protection of encryption
- Confidentiality: The encrypted communication between a user and a trusted host may be subject to a “man-in-the-middle” sniffing attack

Exposure period

- Design: Proper authentication should be included in the system design.
- Design: Use a language which provides an interface to safely handle this exchange.
- Implementation: If use of SSL (or similar) is simply mandated by design and requirements, it is the implementor’s job to properly use the API and all its protections.

Platform

- Languages: Any language which does not provide a framework for key exchange.
- Operating platforms: All

Required resources

Any

Severity

High

Likelihood of exploit

High

Avoidance and mitigation

- Design: Ensure that proper authentication is included in the system design.
- Implementation: Understand and properly implement all checks necessary to ensure the identity of entities involved in encrypted communications.

Discussion

Key exchange without entity authentication may lead to a set of attacks known as “man-in-the-middle” attacks. These attacks take place through the impersonation of a trusted server by a malicious server. If the user skips or ignores the failure of authentication, the server may request authentication information from the user and then use this information with the true server to either sniff the legitimate traffic between the user and host or simply to log in manually with the user’s credentials.

Examples

Many systems have used Diffie-Hellman key exchange without authenticating the entities exchanging keys, leading to man-in-the-middle attacks. Many people using SSL/TLS skip the authentication (often unknowingly).

Related problems

- Failure to follow chain of trust in certificate validation
- Failure to validate host-specific certificate data
- Failure to validate certificate expiration
- Failure to check for certificate revocation

Failure to validate host-specific certificate data

Overview

The failure to validate host-specific certificate data may mean that, while the certificate read was valid, it was not for the site originally requested.

Consequences

- Integrity: The data read from the system vouched for by the certificate may not be from the expected system.
- Authentication: Trust afforded to the system in question — based on the expired certificate — may allow for spoofing or redirection attacks.

Exposure period

- Design: Certificate verification and handling should be performed in the design phase.

Platform

- Language: All
- Operating platform: All

Required resources

Minor trust: Users must attempt to interact with the malicious system.

Severity

High

Likelihood of exploit

High

Avoidance and mitigation

- Design: Check for expired certificates and provide the user with adequate information about the nature of the problem and how to proceed.

Discussion

If the host-specific data contained in a certificate is not checked, it may be possible for a redirection or spoofing attack to allow a malicious host with a valid certificate to provide data, impersonating a trusted host.

While the attacker in question may have a valid certificate, it may simply be a valid certificate for a different site. In order to ensure data integrity, we must check that the certificate is valid and that it pertains to the site that we wish to access.

Examples

```
if (!(cert = SSL_get_peer(certificate(ssl)) || !host)
  foo=SSL_get_veryify_result(ssl);
  if ((X509_V_OK==foo) || X509_V_ERR_SUBJECT_ISSUER_MISMATCH==foo))
```

```
        //do stuff
```

Related problems

- Failure to follow chain of trust in certificate validation
- Failure to validate certificate expiration
- Failure to check for certificate revocation

Failure to validate certificate expiration

Overview

The failure to validate certificate operation may result in trust being assigned to certificates which have been abandoned due to age.

Consequences

- Integrity: The data read from the system vouched for by the expired certificate may be flawed due to malicious spoofing.
- Authentication: Trust afforded to the system in question — based on the expired certificate — may allow for spoofing attacks.

Exposure period

- Design: Certificate expiration handling should be performed in the design phase.

Platform

- Languages: All
- Platforms: All

Required resources

Minor trust: Users must attempt to interact with the malicious system.

Severity

Low

Likelihood of exploit

Low

Avoidance and mitigation

- Design: Check for expired certificates and provide the user with adequate information about the nature of the problem and how to proceed.

Discussion

When the expiration of a certificate is not taken in to account, no trust has necessarily been conveyed through it; therefore, all benefit of certificate is lost.

Examples

```
if (!(cert = SSL_get_peer(certificate(ssl)) || !host)
  foo=SSL_get_veryify_result(ssl);
  if ((X509_V_OK==foo) || (X509_V_ERRCERT_NOT_YET_VALID==foo))
//do stuff
```

Related problems

- Failure to follow chain of trust in certificate validation

- Failure to validate host-specific certificate data
- Key exchange without entity authentication
- Failure to check for certificate revocation
- Using a key past its expiration date

Failure to check for certificate revocation

Overview

If a certificate is used without first checking to ensure it was not revoked, the certificate may be compromised.

Consequences

- Authentication: Trust may be assigned to an entity who is not who it claims to be.
- Integrity: Data from an untrusted (and possibly malicious) source may be integrated.
- Confidentiality: Date may be disclosed to an entity impersonating a trusted entity, resulting in information disclosure.

Exposure period

- Design: Checks for certificate revocation should be included in the design of a system.
- Design: One can choose to use a language which abstracts out this part of authentication and encryption.

Platform

- Languages: Any language which does not abstract out this part of the process
- Operating platforms: All

Required resources

Minor trust: Users must attempt to interact with the malicious system.

Severity

Medium

Likelihood of exploit

Medium

Avoidance and mitigation

- Design: Ensure that certificates are checked for revoked status.

Discussion

The failure to check for certificate revocation is a far more serious flaw than related certificate failures. This is because the use of any revoked certificate is almost certainly malicious. The most common reason for certificate revocation is compromise of the system in question, with the result that no legitimate servers will be using a revoked certificate, unless they are sorely out of sync.

Examples

In C/C++:

```
if (!(cert = SSL_get_peer(certificate(ssl)) || !host)
        ... without a get_verify_results
```

Related problems

- Failure to follow chain of trust in certificate validation
- Failure to validate host-specific certificate data
- Key exchange without entity authentication
- Failure to check for certificate expiration

Failure to encrypt data

Overview

The failure to encrypt data passes up the guarantees of confidentiality, integrity, and accountability that properly implemented encryption conveys.

Consequences

- Confidentiality: Properly encrypted data channels ensure data confidentiality.
- Integrity: Properly encrypted data channels ensure data integrity.
- Accountability: Properly encrypted data channels ensure accountability.

Exposure period

- Requirements specification: Encryption should be a requirement of systems that transmit data.
- Design: Encryption should be designed into the system at the architectural and design phases

Platform

- Languages: Any
- Operating platform: Any

Required resources

Any

Severity

High

Likelihood   of exploit

Very High

Avoidance and mitigation

- Requirements specification: require that encryption be integrated into the system.
- Design: Ensure that encryption is properly integrated into the system design, not simply as a drop-in replacement for sockets.

Discussion

Omitting the use of encryption in any program which transfers data over a network of any kind should be considered on par with delivering the data sent to each user on the local networks of both the sender and receiver.

Worse, this omission allows for the injection of data into a stream of communication between two parties — with no means for the victims to separate valid data from invalid.

In this day of widespread network attacks and password collection sniffers, it is an unnecessary risk to omit encryption from the design of any system which might benefit from it.

Examples

In C:

```
server.sin_family = AF_INET;
hp = gethostbyname(argv[1]);
if (hp==NULL) error("Unknown host");
memcpy( (char *)&server.sin_addr,(char *)hp->h_addr,hp->h_length);
if (argc < 3) port = 80;
else port = (unsigned short)atoi(argv[3]);
server.sin_port = htons(port);
if (connect(sock, (struct sockaddr *)&server, sizeof server) < 0)
         error("Connecting");
...

  while ((n=read(sock,buffer,BUFSIZE-1))!=-1){
          write(dfd,password_buffer,n);
.
.
.
```

In Java:

```
try {
  URL u = new URL("http://www.importantsecretsite.org/");
  HttpURLConnection hu = (HttpURLConnection) u.openConnection();
  hu.setRequestMethod("PUT");
  hu.connect();
  OutputStream os = hu.getOutputStream();
  hu.disconnect();
}
catch (IOException e) { //...
```

Related problems

Not available.

Failure to add integrity check value

Overview

If integrity check values or "checksums" are omitted from a protocol, there is no way of determining if data has been corrupted in transmission.

Consequences

- Integrity: Data that is parsed and used may be corrupted.
- Non-repudiation: Without a checksum it is impossible to determine if any changes have been made to the data after it was sent.

Exposure period

- Design: Checksums are an aspect of protocol design and should be handled there.
- Implementation: Checksums must be properly created and added to the messages in the correct manner to ensure that they are correct when sent.

Platform

- Languages: All
- Platforms: All

Required resources

Network proximity: Some ability to inject messages into a stream, or otherwise corrupt network traffic, would be required to capitalize on this flaw.

Severity

High

Likelihood of exploit

Medium

Avoidance and mitigation

- Design: Add an appropriately sized checksum to the protocol, ensuring that data received may be simply validated before it is parsed and used.
- Implementation: Ensure that the checksums present in the protocol design are properly implemented and added to each message before it is sent.

Discussion

The failure to include checksum functionality in a protocol removes the first application-level check of data that can be used. The end-to-end philosophy of checks states that integrity checks should be performed at the lowest level that they can be completely implemented. Excluding further sanity checks and input validation performed by applications, the protocol's checksum is the most important level of checksum, since it can be performed more completely than at any previous level and takes into account entire messages, as opposed to single packets.

Failure to add this functionality to a protocol specification, or in the implementation of that protocol, needlessly ignores a simple solution for a very significant problem and should never be skipped.

Examples

In C/C++:

```
int r,s;
struct hostent *h;
struct sockaddr_in rserv,lserv;
h=gethostbyname("127.0.0.1");
rserv.sin_family=h->h_addrtype;
memcpy((char *) &rserv.sin_addr.s_addr, h->h_addr_list[0]
  ,h->h_length);
rserv.sin_port= htons(1008);
s = socket(AF_INET,SOCK_DGRAM,0);

lserv.sin_family = AF_INET;
lserv.sin_addr.s_addr = htonl(INADDR_ANY);
lserv.sin_port = htons(0);

r = bind(s, (struct sockaddr *) &lserv,sizeof(lserv));
sendto(s,important_data,strlen(improtant_data)+1,0
   ,(struct sockaddr *) &rserv, sizeof(rserv));
```

In Java:

```
while(true) {
  DatagramPacket rp=new DatagramPacket(rData,rData.length);

  outSock.receive(rp);
  String in = new String(p.getData(),0, rp.getLength());
  InetAddress IPAddress = rp.getAddress();
  int port = rp.getPort();

    out = secret.getBytes();
    DatagramPacket sp =new DatagramPacket(out,out.length,
      IPAddress, port);
    outSock.send(sp);
  }
}
```

Related problems

- Failure to check integrity check value

Failure to check integrity check value

Overview

If integrity check values or “checksums” are not validated before messages are parsed and used, there is no way of determining if data has been corrupted in transmission.

Consequences

- Authentication: Integrity checks usually use a secret key that helps authenticate the data origin. Skipping integrity checking generally opens up the possibility that new data from an invalid source can be injected.
- Integrity: Data that is parsed and used may be corrupted.
- Non-repudiation: Without a checksum check, it is impossible to determine if any changes have been made to the data after it was sent.

Exposure period

- Implementation: Checksums must be properly checked and validated in the implementation of message receiving.

Platform

- Languages: All
- Operating platforms: All

Required resources

Any

Severity

High

Likelihood of exploit

Medium

Avoidance and mitigation

- Implementation: Ensure that the checksums present in messages are properly checked in accordance with the protocol specification before they are parsed and used.

Discussion

The failure to validate checksums before use results in an unnecessary risk that can easily be mitigated with very few lines of code. Since the protocol specification describes the algorithm used for calculating the checksum, it is a simple matter of implementing the calculation and verifying that the calculated checksum and the received checksum match.

If this small amount of effort is skipped, the consequences may be far greater.

Examples

In C/C++:

```
sd = socket(AF_INET, SOCK_DGRAM, 0);
serv.sin_family = AF_INET;
serv.sin_addr.s_addr = htonl(INADDR_ANY);
servr.sin_port = htons(1008);
bind(sd, (struct sockaddr *) & serv, sizeof(serv));
while (1) {
  memset(msg, 0x0, MAX_MSG);
  clilen = sizeof(cli);
  if (inet_ntoa(cli.sin_addr)==...)
  n = recvfrom(sd, msg, MAX_MSG, 0,
              (struct sockaddr *) & cli, &clilen);
}
```

In Java:

```
while(true) {
  DatagramPacket packet
    = new DatagramPacket(data,data.length,IPAddress, port);
  socket.send(sendPacket);
}
```

Related problems

- Failure to add integrity check value

Use of hard-coded password

Overview

The use of a hard-coded password increases the possibility of password guessing tremendously.

Consequences

- Authentication: If hard-coded passwords are used, it is almost certain that malicious users will gain access through the account in question.

Exposure period

- Design: For both front-end to back-end connections and default account settings, alternate decisions must be made at design time.

Platform

- Languages: All
- Operating platforms: All

Required resources

Knowledge of the product or access to code.

Severity

High

Likelihood of exploit

Very high

Avoidance and mitigation

- Design (for default accounts): Rather than hard code a default username and password for first time logins, utilize a “first login” mode which requires the user to enter a unique strong password.
- Design (for front-end to back-end connections): Three solutions are possible, although none are complete. The first suggestion involves the use of generated passwords which are changed automatically and must be entered at given time intervals by a system administrator. These passwords will be held in memory and only be valid for the time intervals. Next, the passwords used should be limited at the back end to only performing actions valid to for the front end, as opposed to having full access. Finally, the messages sent should be tagged and checksummed with time sensitive values so as to prevent replay style attacks.

Discussion

The use of a hard-coded password has many negative implications — the most significant of these being a failure of authentication measures under certain circumstances.

On many systems, a default administration account exists which is set to a simple default password which is hard-coded into the program or device. This hard-coded password is the same for each device or sys-tem of this type and often is not changed or disabled by end users. If a malicious user comes across a device of this kind, it is a simple matter of looking up the default password (which is freely available and public on the internet) and logging in with complete access.

In systems which authenticate with a back-end service, hard-coded passwords within closed source or drop-in solution systems require that the back-end service use a password which can be easily discovered. Client-side systems with hard-coded passwords propose even more of a threat, since the extraction of a password from a binary is exceedingly simple.

Examples

In C\C++:

```
int VerifyAdmin(char *password) {

  if (strcmp(password, “Mew!”)) {
    printf(“Incorrect Password!\n”);
    return(0)
  }

  printf(“Entering Diagnostic Mode…\n”);
  return(1);
}
```

In Java:

```
int VerifyAdmin(String password) {

  if (passwd.Eqauls(“Mew!”)) {
    return(0)
  }
//Diagnostic Mode
  return(1);
}
```

Every instance of this program can be placed into diagnostic mode with the same password. Even worse is the fact that if this program is distributed as a binary-only distribution, it is very difficult to change that password or disable this “functionality.”

Related problems

- Use of hard-coded cryptographic key
- Storing passwords in a recoverable format

Use of hard-coded cryptographic key

Overview

The use of a hard-coded cryptographic key tremendously increases the possibility that encrypted data may be recovered

Consequences

- Authentication: If hard-coded cryptographic keys are used, it is almost certain that malicious users will gain access through the account in question.

Exposure period

- Design: For both front-end to back-end connections and default account settings, alternate decisions must be made at design time.

Platform

- Languages: All
- Operating platforms: All

Required resources

Any

Severity

High

Likelihood of exploit

High

Avoidance and mitigation

- Design: Prevention schemes mirror that of hard-coded password storage.

Discussion

The main difference between the use of hard-coded passwords and the use of hard-coded cryptographic keys is the false sense of security that the former conveys. Many people believe that simply hashing a hard-coded password before storage will protect the information from malicious users. However, many hashes are reversible (or at least vulnerable to brute force attacks) — and further, many authentication protocols simply request the hash itself, making it no better than a password.

Examples

In C\C++:

```
int VerifyAdmin(char *password) {
  if (strcmp(password,”68af404b513073584c4b6f22b6c63e6b”)) {
    printf(“Incorrect Password!\n”);
    return(0)
  }
```

```
            printf(“Entering Diagnostic Mode…\n”);
            return(1);
          }
```

In Java:

```
          int VerifyAdmin(String password) {

            if (passwd.Eqauls(“68af404b513073584c4b6f22b6c63e6b”)) {
              return(0)
            }
          //Diagnostic Mode
            return(1);
          }
```

Related problems

- Use of hard-coded password

Storing passwords in a recoverable format

Overview

The storage of passwords in a recoverable format makes them subject to password reuse attacks by malicious users. If a system administrator can recover the password directly — or use a brute force search on the information available to him —, he can use the password on other accounts.

Consequences

- Confidentiality: User's passwords may be revealed.
- Authentication: Revealed passwords may be reused elsewhere to impersonate the users in question.

Exposure period

- Design: The method of password storage and use is often decided at design time.
- Implementation: In some cases, the decision of algorithms for password encryption or hashing may be left to the implementers.

Platform

- Languages: All
- Operating platforms: All

Required resources

Access to read stored password hashes

Severity

Medium to High

Likelihood   of exploit

Very High

Avoidance and mitigation

- Design / Implementation: Ensure that strong, non-reversible encryption is used to protect stored passwords.

Discussion

The use of recoverable passwords significantly increases the chance that passwords will be used maliciously. In fact, it should be noted that recoverable encrypted passwords provide no significant benefit over plain-text passwords since they are subject not only to reuse by malicious attackers but also by malicious insiders.

Examples

In C\C++:

```
int VerifyAdmin(char *password) {
  if (strcmp(compress(password), compressed_password)) {
    printf(“Incorrect Password!\n”);
    return(0)
  }

  printf(“Entering Diagnostic Mode…\n”);
  return(1);
}
```

In Java:

```
int VerifyAdmin(String password) {

  if (passwd.Eqauls(compress((compressed_password)) {
    return()0)
  }
//Diagnostic Mode
  return(1);
}
```

Related problems

- Use of hard-coded passwords

Trusting self-reported IP address

Overview

The use of IP addresses as authentication is flawed and can easily be spoofed by malicious users.

Consequences

- Authentication: Malicious users can fake authentication information, impersonating any IP address

Exposure period

- Design: Authentication methods are generally chosen during the design phase of development.

Platform

- Languages: All
- Operating platforms: All

Required resources

Any

Severity

High

Likelihood of exploit

High

Avoidance and mitigation

- Design: Use other means of identity verification that cannot be simply spoofed.

Discussion

As IP addresses can be easily spoofed, they do not constitute a valid authentication mechanism. Alternate methods should be used if significant authentication is necessary.

Examples

In C/C++:

```
sd = socket(AF_INET, SOCK_DGRAM, 0);
serv.sin_family = AF_INET;
serv.sin_addr.s_addr = htonl(INADDR_ANY);
servr.sin_port = htons(1008);
bind(sd, (struct sockaddr *) & serv, sizeof(serv));
while (1) {
  memset(msg, 0x0, MAX_MSG);
  clilen = sizeof(cli);
  if (inet_ntoa(cli.sin_addr)==...)
  n = recvfrom(sd, msg, MAX_MSG, 0,
              (struct sockaddr *) & cli, &clilen);
}
```

In Java:

```
while(true) {
  DatagramPacket rp=new DatagramPacket(rData,rData.length);

  outSock.receive(rp);
  String in = new String(p.getData(),0, rp.getLength());
  InetAddress IPAddress = rp.getAddress();
  int port = rp.getPort();

  if ((rp.getAddress()==...) && (in==...)){
    out = secret.getBytes();
    DatagramPacket sp =new DatagramPacket(out,out.length,
      IPAddress, port);
    outSock.send(sp);
  }
}
```

Related problems

- Trusting self-reported DNS name
- Using the referer field for authentication

Trusting self-reported DNS name

Overview

The use of self-reported DNS names as authentication is flawed and can easily be spoofed by malicious users.

Consequences

Authentication: Malicious users can fake authentication information by providing false DNS information.

Exposure period

- Design: Authentication methods are generally chosen during the design phase of development.

Platform

- Languages: All
- Operating platforms: All

Required resources

Any

Severity

High

Likelihood   of exploit

High

Avoidance and mitigation

- Design: Use other means of identity verification that cannot be simply spoofed.

Discussion

As DNS names can be easily spoofed or mis-reported, they do not constitute a valid authentication mechanism. Alternate methods should be used if the significant authentication is necessary.

In addition, DNS name resolution as authentication would — even if it was a valid means of authentication — imply a trust relationship with the DNS servers used, as well as all of the servers they refer to.

Examples

In C/C++:

```
sd = socket(AF_INET, SOCK_DGRAM, 0);
serv.sin_family = AF_INET;
serv.sin_addr.s_addr = htonl(INADDR_ANY);
servr.sin_port = htons(1008);
bind(sd, (struct sockaddr *) & serv, sizeof(serv));
while (1) {
  memset(msg, 0x0, MAX_MSG);
```

```
            clilen = sizeof(cli);
            h=gethostbyname(inet_ntoa(cliAddr.sin_addr));
            if (h->h_name==...)
            n = recvfrom(sd, msg, MAX_MSG, 0,
                        (struct sockaddr *) & cli, &clilen);
          }
```

In Java:

```
          while(true) {
            DatagramPacket rp=new DatagramPacket(rData,rData.length);

            outSock.receive(rp);
            String in = new String(p.getData(),0, rp.getLength());
            InetAddress IPAddress = rp.getAddress();
            int port = rp.getPort();

            if ((rp.getHostName()==...) && (in==...)){
              out = secret.getBytes();
              DatagramPacket sp =new DatagramPacket(out,out.length,
                IPAddress, port);
              outSock.send(sp);
            }
          }
```

Related problems

- Trusting self-reported IP address
- Using referrer field for authentication

Using referrer field for authentication

Overview

The referrer field in HTTP requests can be easily modified and, as such, is not a valid means of message integrity checking.

Consequences

- Authorization: Actions, which may not be authorized otherwise, can be carried out as if they were validated by the server referred to.
- Accountability: Actions may be taken in the name of the server referred to.

Exposure period

- Design: Authentication methods are generally chosen during the design phase of development.

Platform

- Languages: All
- Operating platforms: All

Required resources

Any

Severity

High

Likelihood   of exploit

Very High

Avoidance and mitigation

- Design: Use other means of authorization that cannot be simply spoofed.

Discussion

The referrer field in HTML requests can be simply modified by malicious users, rendering it useless as a means of checking the validity of the request in question. In order to usefully check if a given action is authorized, some means of strong authentication and method protection must be used.

Examples

In C/C++:

```
sock= socket(AF_INET, SOCK_STREAM, 0);
...
bind(sock, (struct sockaddr *)&server, len)
...
while (1)
newsock=accept(sock, (struct sockaddr *)&from, &fromlen);
pid=fork();
```

```
if (pid==0) {
  n = read(newsock,buffer,BUFSIZE);
...
if (buffer+...==Referer: http://www.foo.org/dsaf.html)
//do stuff
```

In Java:

```
public class httpd extends Thread{
  Socket cli;
  public httpd(Socket serv){
    cli=serv;
    start();
  }
  public static void main(String[]a){
  ...
  ServerSocket serv=new ServerSocket(8181);
  for(;;){
    new h(serv.accept());
  ...
   public void run(){
     try{
       BufferedReader reader
         =new BufferedReader(new InputStreamReader(cli.getInputStream()));
       //if i contains a the proper referer.

      DataOutputStream o=
         new DataOutputStream(c.getOutputStream());
      ...
```

Related problems

- Trusting self-reported IP address
- Using the referer field for authentication

Using a broken or risky cryptographic algorithm

Overview

The use of a broken or risky cryptographic algorithm is an unnecessary risk that may result in the disclosure of sensitive information.

Consequences

- Confidentiality: The confidentiality of sensitive data may be compromised by the use of a broken or risky cryptographic algorithm.
- Integrity: The integrity of sensitive data may be compromised by the use of a broken or risky cryptographic algorithm.
- Accountability: Any accountability to message content preserved by cryptography may be subject to attack.

Exposure period

- Design: The decision as to what cryptographic algorithm to utilize is generally made at design time.

Platform

- Languages: All
- Operating platforms: All

Required resources

Any

Severity

High

Likelihood of exploit

Medium to High

Avoidance and mitigation

- Design: Use a cryptographic algorithm that is currently considered to be strong by experts in the field.

Discussion

Since the state of cryptography advances so rapidly, it is common to find algorithms, which previously were considered to be safe, currently considered unsafe. In some cases, things are discovered, or processing speed increases to the degree that the cryptographic algorithm provides little more benefit than the use of no cryptography at all.

Examples

In C/C++:

```
EVP_des_ecb();
```

In Java:

```
Cipher des=Cipher.getInstance("DES...);
des.initEncrypt(key2);
```

Related problems

- Failure to encrypt data

Using password systems

Overview

The use of password systems as the primary means of authentication may be subject to several flaws or shortcomings, each reducing the effectiveness of the mechanism.

Consequences

- Authentication: The failure of a password authentication mechanism will almost always result in attackers being authorized as valid users.

Exposure period

- Design: The period of development in which authentication mechanisms and their protections are devised is the design phase.

Platform

- Languages: All
- Operating platforms: All

Required resources

Any

Severity

High

Likelihood   of exploit

Very High

Avoidance and mitigation

- Design: Use a zero-knowledge password protocol, such as SRP.
- Design: Ensure that passwords are sorted safely and are not reversible.
- Design: Implement password aging functionality that requires passwords be changed after a certain point.
- Design: Use a mechanism for determining the strength of a password and notify the user of weak password use.
- Design: Inform the user of why password protections are in place, how they work to protect data integrity, and why it is important to heed their warnings.

Discussion

Password systems are the simplest and most ubiquitous authentication mechanisms. However, they are subject to such well known attacks, and such frequent compromise that their use in the most simple implementation is not practical. In order to protect password systems from compromise, the following should be noted:

- Passwords should be stored safely to prevent insider attack and to ensure that — if a system is compromised — the passwords are not retrievable. Due to password reuse, this information may be useful in the compromise of other systems these users work with. In order to protect these passwords, they should be stored encrypted, in a non-reversible state, such that the original text password cannot be extracted from the stored value.
- Password aging should be strictly enforced to ensure that passwords do not remain unchanged for long periods of time. The longer a password remains in use, the higher the probability that it has been compromised. For this reason, passwords should require refreshing periodically, and users should be informed of the risk of passwords which remain in use for too long.
- Password strength should be enforced intelligently. Rather than restrict passwords to specific content, or specific length, users should be encouraged to use upper and lower case letters, numbers, and symbols in their passwords. The system should also ensure that no passwords are derived from dictionary words.

Examples

```
unsigned char *check_passwd(char *plaintext){
        ctext=simple_digest("sha1",plaintext,strlen(plaintext)...);
        if (ctext==secret_password())
          // Log me in
}
```

In Java:

```
String plainText = new String(plainTextIn)
MessageDigest encer = MessageDigest.getInstance("SHA");
encer.update(plainTextIn);
byte[] digest = password.digest();
if (digest==secret_password())
//log me in
```

Related problems

- Using single-factor authentication

Using single-factor authentication

Overview

The use of single-factor authentication can lead to unnecessary risk of compromise when compared with the benefits of a dual-factor authentication scheme.

Consequences

- Authentication: If the secret in a single-factor authentication scheme gets compromised, full authentication is possible.

Exposure period

- Design: Authentication methods are determined at design time.

Platform

- Languages: All
- Operating platform: All

Required resources

Any

Severity

High

Likelihood of exploit

High

Avoidance and mitigation

- Design: Use multiple independent authentication schemes, which ensures that — if one of the methods is compromised — the system itself is still likely safe from compromise.

Discussion

While the use of multiple authentication schemes is simply piling on more complexity on top of authentication, it is inestimably valuable to have such measures of redundancy.

The use of weak, reused, and common passwords is rampant on the internet. Without the added protection of multiple authentication schemes, a single mistake can result in the compromise of an account. For this reason, if multiple schemes are possible and also easy to use, they should be implemented and required.

Examples

In C:

```
unsigned char *check_passwd(char *plaintext){
        ctext=simple_digest("sha1",plaintext,strlen(plaintext)...);
        if (ctext==secret_password())
          // Log me in
        }
```

In Java:

```
String plainText = new String(plainTextIn)
MessageDigest encer = MessageDigest.getInstance("SHA");
encer.update(plainTextIn);
byte[] digest = password.digest();
if (digest==secret_password())
  //log me in
```

Related problems

- Using password systems

Not allowing password aging

Overview

If no mechanism is in place for managing password aging, users will have no incentive to update passwords in a timely manner.

Consequences

- Authentication: As passwords age, the probability that they are compromised grows.

Exposure period

- Design: Support for password aging mechanisms must be added in the design phase of development.

Platform

- Languages: All
- Operating platforms: All

Required resources

Any

Severity

Medium

Likelihood of exploit

Very Low

Avoidance and mitigation

- Design: Ensure that password aging functionality is added to the design of the system, including an alert previous to the time the password is considered obsolete, and useful information for the user concerning the importance of password renewal, and the method.

Discussion

The recommendation that users change their passwords regularly and do not reuse passwords is universal among security experts. In order to enforce this, it is useful to have a mechanism that notifies users when passwords are considered old and that requests that they replace them with new, strong passwords.

In order for this functionality to be useful, however, it must be accompanied with documentation which stresses how important this practice is and which makes the entire process as simple as possible for the user.

Examples

- A common example is not having a system to terminate old employee accounts.
- Not having a system for enforcing the changing of passwords every certain period.

Related problems

- Using password systems
- Allowing password aging
- Using a key past its expiration date

Allowing password aging

Overview

Allowing password aging to occur unchecked can result in the possibility of diminished password integrity.

Consequences

- Authentication: As passwords age, the probability that they are compromised grows.

Exposure period

- Design: Support for password aging mechanisms must be added in the design phase of development.

Platform

- Languages: All
- Operating platforms: All

Required resources

Any

Severity

Medium

Likelihood of exploit

Very Low

Avoidance and mitigation

- Design: Ensure that password aging is limited so that there is a defined maximum age for passwords and so that the user is notified several times leading up to the password expiration.

Discussion

Just as neglecting to include functionality for the management of password aging is dangerous, so is allowing password aging to continue unchecked. Passwords must be given a maximum life span, after which a user is required to update with a new and different password.

Examples

- A common example is not having a system to terminate old employee accounts.
- Not having a system for enforcing the changing of passwords every certain period.

Related problems

- Not allowing for password aging

Reusing a nonce, key pair in encryption

Overview

Nonces should be used for the present occasion and only once.

Consequences

- Authentication: Potentially a replay attack, in which an attacker could send the same data twice, could be crafted if nonces are allowed to be reused. This could allow a user to send a message which masquerades as a valid message from a valid user.

Exposure period

- Design: Mitigating technologies such as safe string libraries and container abstractions could be introduced.
- Implementation: Many traditional techniques can be used to create a new nonce from different sources.
- Implementation: Reusing nonces nullifies the use of nonces.

Platform

- Languages: Any
- Operating platforms: Any

Required resources

Any

Severity

High

Likelihood   of exploit

High

Avoidance and mitigation

- Requirements specification: The choice could be made to use a language that is not susceptible to these issues.
- Implementation: Refuse to reuse nonce values.
- Implementation: Use techniques such as requiring incrementing, time based and/or challenge response to assure uniqueness of nonces.

Discussion

Nonces, are often bundled with a key in a communication exchange to produce a new session key for each exchange.

Examples

In C/C++:

```
#include <openssl/sha.h>
#include <stdio.h>
#include <string.h>
#include <memory.h>

int main(){
  char *paragraph = NULL;
  char *data = NULL;
  char *nonce = “bad“;
  char *password = “secret”;

  parsize=strlen(nonce)+strlen(password);
  paragraph=(char*)malloc(para_size);
  strncpy(paragraph,nonce,strlen(nonce));
  strcpy(paragraph,password,strlen(password));

  data=(unsigned char*)malloc(20);
  SHA1((const unsigned char*)paragraph,parsize,(unsigned char*)data);

  free(paragraph);
  free(data);
//Do something with data//
  return 0;
}
```

In Java:

```
String command = new String(“some command to execute”)
MessageDigest nonce = MessageDigest.getInstance("SHA");
nonce.update(String.valueOf(“bad nonce”);
byte[] nonce = nonce.digest();

MessageDigest password = MessageDigest.getInstance("SHA");
password.update(nonce + “secretPassword”);
byte[] digest = password.digest();
//do somethign with digest//
```

Related problems

Not available.

Using a key past its expiration date

Overview

The use of a cryptographic key or password past its expiration date diminishes its safety significantly.

Consequences

- Authentication: The cryptographic key in question may be compromised, providing a malicious user with a method for authenticating as the victim.

Exposure period

- Design: The handling of key expiration should be considered during the design phase — largely pertaining to user interface design.
- Run time: Users are largely responsible for the use of old keys.

Platform

- Languages: All
- Platforms: All

Required resources

Any

Severity

Low

Likelihood   of exploit

Low

Avoidance and mitigation

- Design: Adequate consideration should be put in to the user interface in order to notify users previous to the key's expiration, to explain the importance of new key generation and to walk users through the process as painlessly as possible.
- Run time: Users must heed warnings and generate new keys and passwords when they expire.

Discussion

While the expiration of keys does not necessarily ensure that they are compromised, it is a significant concern that keys which remain in use for prolonged periods of time have a decreasing probability of integrity.

For this reason, it is important to replace keys within a period of time proportional to their strength.

Examples

In C/C++:

```
if (!(cert = SSL_get_peer(certificate(ssl)) || !host)
  foo=SSL_get_veryify_result(ssl);
  if ((X509_V_OK==foo) || (X509_V_ERRCERT_NOT_YET_VALID==foo))
//do stuff
```

Related problems

- Failure to check for certificate expiration

Not using a random IV with CBC mode

Overview

Not using a random initialization vector with Cipher Block Chaining (CBC) Mode causes algorithms to be susceptible to dictionary attacks.

Consequences

- Confidentiality: If the CBC is not properly initialized, data which is encrypted can be compromised and therefore be read.
- Integrity: If the CBC is not properly initialized, encrypted data could be tampered with in transfer or if it accessible.
- Accountability: Cryptographic based authentication systems could be defeated.

Exposure period

- Implementation: Many logic errors can lead to this condition if multiple data streams have a common beginning sequences.

Platform

- Languages: Any
- Operating platforms: Any

Required resources

.Any

Severity

High

Likelihood   of exploit

Medium

Avoidance and mitigation

- Integrity: It is important to properly initialize CBC operating block ciphers or there use is lost.

Discussion

CBC is the most commonly used mode of operation for a block cipher. It solves electronic code book's dictionary problems by XORing the ciphertext with plaintext. If it used to encrypt multiple data streams, dictionary attacks are possible, provided that the streams have a common beginning sequence.

Examples

In C/C++:

```
#include <openssl/evp.h>

EVP_CIPHER_CTX ctx;
```

```
char key[EVP_MAX_KEY_LENGTH];
char iv[EVP_MAX_IV_LENGTH];

RAND_bytes(key, b);
memset(iv,0,EVP_MAX_IV_LENGTH);
EVP_EncryptInit(&ctx,EVP_bf_cbc(), key,iv);
```

In Java:

```
public class SymmetricCipherTest {
   public static void main()  {
      byte[] text =”Secret".getBytes();
      byte[] iv ={0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00};

      KeyGenerator kg = KeyGenerator.getInstance("DES");
      kg.init(56);
      SecretKey key = kg.generateKey();

      Cipher cipher = Cipher.getInstance("DES/ECB/PKCS5Padding");
      IvParameterSpec ips = new IvParameterSpec(iv);
      cipher.init(Cipher.ENCRYPT_MODE, key, ips);
      return cipher.doFinal(inpBytes);
   }
 }
```

Related problems

Not available.

Failure to protect stored data from modification

Overview

Data should be protected from direct modification.

Consequences

- Integrity: The object could be tampered with.

Exposure period

- Design through Implementation: At design time it is important to reduce the total amount of accessible data.
- Implementation: Most implementation level issues come from a lack of understanding of the language modifiers.

Platform

- Languages: Java, C++
- Operating platforms: Any

Required resources

Any

Severity

Medium

Likelihood of exploit

Medium

Avoidance and mitigation

- Design through Implementation: Use private members, and class accessor methods to their full benefit. This is the recommended mitigation. Make all public members private, and — if external access is necessary — use accessor functions to do input validation on all values.
- Implementation: Data should be private, static, and final whenever possible This will assure that your code is protected by instantiating early, preventing access and preventing tampering.
- Implementation: Use sealed classes. Using sealed classes protects object-oriented encapsulation paradigms and therefore protects code from being extended in unforeseen ways.
- Implementation: Use class accessor methods to their full benefit. Use the accessor functions to do input validation on all values intended for private values.

Discussion

One of the main advantages of object-oriented code is the ability to limit access to fields and other resources by way of accessor functions. Utilize accessor functions to make sure your objects are well-formed.

Final provides security by only allowing non-mutable objects to be changed after being set. However, only objects which are not extended can be made final.

Examples

In C++:

```
public:
  int someNumberPeopleShouldntMessWith;
```

In Java:

```
private class parserProg {
    public stringField;
}
```

Another set of Examples are:

In C/C++:

```
private:
  int someNumber;

public:
  void writeNum(int newNum) {
    someNumber = newNum;
  }
```

In Java:

```
public class eggCorns {
   private String acorns;
   public void misHear(String name){
      acorns=name;
   }
}
```

Related problems

Not available.

Failure to provide confidentiality for stored data

Overview

Non-final public fields should be avoided, if possible, as the code is easily tamperable.

Consequences

- Integrity: The object could potentially be tampered with.
- Confidentiality: The object could potentially allow the object to be read.

Exposure period

- Implementation: This flaw is a simple logic issue, introduced entirely at implementation time.

Platform

- Languages: Java, C++
- Operating platforms: Any

Required resources

Any

Severity

Medium

Likelihood   of exploit

High

Avoidance and mitigation

- Implementation: Make any non-final field private.

Discussion

If a field is non-final and public, it can be changed once their value is set by any function which has access to the class which contains the field.

Examples

In C++:

```
public int password r = 45;
```

In Java:

```
public String r = new String("My Password");
```

Now this field is readable from any function and can be changed by any function.

Related problems

Not available.

Category 5: General Logic Errors

This section introduces the vulnerability Problem Types organized under the problem type "general logic errors."

Ignored function return value

Overview

If a functions return value is not checked, it could have failed without any warning.

Consequences

- Integrity: The data which was produced as a result of a function could be in a bad state.

Exposure period

Implementation: This flaw is a simple logic issue, introduced entirely at implementation time.

Platform

- Languages: C or C++
- Operating platforms: Any

Required resources

Any

Severity

Medium

Likelihood of exploit

Low

Avoidance and mitigation

- Implementation: Check all functions which return a value
- Implementation: When designing any function make sure you return a value or throw an exception in case of an error
- discussion

Important and common functions will return some value about the success of its actions. This will alert the program whether or not to handle any errors caused by that function

Example

In C/C++:

```
malloc(sizeof(int)*4);
```

In Java:

Although some Java members may use return values to state there status, it is preferable to use exceptions.

Related problems

Not available.

Missing parameter

Overview

If too few arguments are sent to a function, the function will still pop the expected number of arguments from the stack. Potentially, a variable number of arguments could be exhausted in a function as well.

Consequences

- Authorization: There is the potential for arbitrary code execution with privileges of the vulnerable program if function parameter list is exhausted.
- Availability: Potentially a program could fail if it needs more arguments then are available.

Exposure period

- Implementation: This is a simple logical flaw created at implementation time.

Platform

- Languages: C or C++
- Operating platforms: Any

Required resources

Any

Severity

High

Likelihood   of exploit

High

Avoidance and mitigation

- Implementation: Forward declare all functions. This is the recommended solution. Properly forward declaration of all used functions will result in a compiler error if too few arguments are sent to a function.

Discussion

This issue can be simply combated with the use of proper build process.

Examples

In C or C++:

```
foo_funct(one, two);
…
void foo_funct(int one, int two, int three) {
  printf(“1) %d\n2) %d\n3) %d\n”, one, two, three);
}
```

This can be exploited to disclose information with no work whatsoever. In fact, each time this function is run, it will print out the next 4 bytes on the stack after the two numbers sent to it.

Another example in C/C++ is:

```
void some_function(int foo, ...) {
    int a[3], i;
    va_list ap;

    va_start(ap, foo);
    for (i = 0;  i < sizeof(a) / sizeof(int);  i++)
        a[i] = va_arg(ap, int);
    va_end(ap);
}

int main(int argc, char *argv[]) {
    some_function(17, 42);
}
```

Related problems

Not available.

Misinterpreted function return value

Overview

If a function’s return value is not properly checked, the function could have failed without proper acknowledgement.

Consequences

- Integrity: The data — which was produced as a result of an improperly checked return value of a function — could be in a bad state.

Exposure period

- Requirements specification: The choice could be made to use a language that uses exceptions rather than return values to handle status.
- Implementation: Many logic errors can lead to this condition. It can be exacerbated by lack, or misuse, of mitigating technologies.

Platform

- Languages: C or C++
- Operating platforms: Any

Required resources

Any

Severity

Medium

Likelihood   of exploit

Low

Avoidance and mitigation

- Requirements specification: Use a language or compiler that uses exceptions and requires the catching of those exceptions.
- Implementation: Properly check all functions which return a value.
- Implementation: When designing any function make sure you return a value or throw an exception in case of an error.

discussion

Important and common functions will return some value about the success of its actions. This will alert the program whether or not to handle any errors caused by that function.

Examples

In C/C++

```
if (malloc(sizeof(int*4) < 0 )
  perror(“Failure”); //should have checked if the call returned 0
```

Related problems

Not available.

Uninitialized variable

Overview

Using the value of an unitialized variable is not safe.

Consequences

- Integrity: Initial variables usually contain junk, which can not be trusted for consistency. This can cause a race condition if a lock variable check passes when it should not.
- Authorization: Strings which do are not initialized are especially dangerous, since many functions expect a null at the end — and only at the end — of a string.

Exposure period

- Implementation: Use of unitialized variables is a logical bug.
- Requirements specification: The choice could be made to use a language that is not susceptible to these issues.
- Design: Mitigating technologies such as safe string libraries and container abstractions could be introduced.

Platform

Languages: C/C++

Operating platforms: Any

Required resources

Any

Severity

High

Likelihood of exploit

High

Avoidance and mitigation

- Implementation: Assign all variables to an initial variable.
- Pre-design through Build: Most compilers will complain about the use of unitialized variables if warnings are turned on.
- Requirements specification: The choice could be made to use a language that is not susceptible to these issues.
- Design: Mitigating technologies such as safe string libraries and container abstractions could be introduced.

Discussion

Before variables are initialized, they generally contain junk data of what was left in the memory that the variable takes up. This data is very rarely useful, and it is generally advised to pre-initialize variables or set them to their first values early.

If one forget — in the C language — to initialize, for example a char *, many of the simple string libraries may often return incorrect results as they expecting the null termination to be at the end of a string.

Examples

In C\C++, or Java:

```
int foo;
void bar(){
  if (foo==0) /.../
    /../
 }
```

Related problems

Not available.

Duplicate key in associative list (alist)

Overview

Associative lists should always have unique keys, since having non-unique keys can often be mistaken for an error.

Consequences

Unspecified.

Exposure period

- Design: The use of a safe data structure could be used.

Platform

- Languages: Although *alists* generally are used only in languages like Common Lisp — due to the functionality overlap with hash tables — an *alist* could appear in a language like C or C++.
- Operating platforms: Any

Required resources

Any

Severity

Medium

Likelihood of exploit

Low

Avoidance and mitigation

- Design: Use a hash table instead of an *alist*.
- Design: Use an *alist* which checks the uniqueness of hash keys with each entry before inserting the entry.

Discussion

A duplicate key entry — if the *alist* is designed properly — could be used as a constant time replace function. However, duplicate key entries could be inserted by mistake. Because of this ambiguity, duplicate key entries in an association list are not recommended and should not be allowed.

Examples

In Python:

```
alist = []
while (foo()):
  #now assume there is a string data with a key basename
  queue.append(basename,data)
queue.sort()
```

Since basename is not necessarily unique, this may not sort how one would like it to be.

Related problems

Not available.

Deletion of data-structure sentinel

Overview

The accidental deletion of a data structure sentinel can cause serious programing logic problems.

Consequences

- Availability: Generally this error will cause the data structure to not work properly.
- Authorization: If a control character, such as NULL is removed, one may cause resource access control problems.

Exposure period

- Requirements specification: The choice could be made to use a language that is not susceptible to these issues.
- Design: Mitigating technologies such as safe-string libraries and container abstractions could be introduced.
- Implementation: Many logic errors can lead to this condition. It can be exacerbated by lack of or misuse of mitigating technologies.

Platform

- Languages: C, C++, Fortran, Assembly
- Operating platforms: All, although partial preventative measures may be deployed depending on environment.

Required resources

Any

Severity

Very High

Likelihood of exploit

High to Very High

Avoidance and mitigation

- Pre-design: Use a language or compiler that performs automatic bounds checking.
- Design: Use an abstraction library to abstract away risky APIs. Not a complete solution.
- Pre-design through Build: Compiler-based canary mechanisms such as StackGuard, ProPolice and the Microsoft Visual Studio / GS flag. Unless this provides automatic bounds checking, it is not a complete solution.
- Operational: Use OS-level preventative functionality. Not a complete solution.

Discussion

Often times data-structure sentinels are used to mark structure of the data structure. A common example of this is the null character at the end of strings. Another common example is linked lists which may contain a sentinel to mark the end of the list.

It is, of course, dangerous to allow this type of control data to be easily accessible. Therefore, it is important to protect from the deletion or modification outside of some wrapper interface which provides safety.

Examples

In C/C++:

```
char *foo;
int counter;
foo=malloc(sizeof(char)*10);
for (counter=0;counter!=14;counter++){
  foo[counter]='a';
  printf("%s\n",foo);
}
```

Related problems

Not available.

Addition of data-structure sentinel

Overview

The accidental addition of a data-structure sentinel can cause serious programing logic problems.

Consequences

- Availability: Generally this error will cause the data structure to not work properly by truncating the data.

Exposure period

- Requirements specification: The choice could be made to use a language that is not susceptible to these issues.
- Design: Mitigating technologies such as safe string libraries and container abstractions could be introduced.
- Implementation: Many logic errors can lead to this condition. It can be exacerbated by lack of or misuse of mitigating technologies.

Platform

- Languages: C, C++, Fortran, Assembly
- Operating platforms: All, although partial preventative measures may be deployed depending on environment.

Required resources

Any

Severity

Very High

Likelihood of exploit

High to Very High

Avoidance and mitigation

- Pre-design: Use a language or compiler that performs automatic bounds checking.
- Design: Use an abstraction library to abstract away risky APIs. Not a complete solution.
- Pre-design through Build: Compiler-based canary mechanisms such as StackGuard, ProPolice, and Microsoft Visual Studio / GS flag. Unless this provides automatic bounds checking, it is not a complete solution.
- Operational: Use OS-level preventative functionality. Not a complete solution.

Discussion

Data-structure sentinels are often used to mark structure of the data structure. A common example of this is the null character at the end of strings. Another common example is linked lists which may contain a sentinel to mark the end of the list.

It is, of course dangerous, to allow this type of control data to be easily accessible. Therefore, it is important to protect from the addition or modification outside of some wrapper interface which provides safety.

By adding a sentinel, one potentially could cause data to be truncated early.

Examples

In C/C++:

```
char *foo;
foo=malloc(sizeof(char)*4);
foo[0]='a';
foo[1]='a';
foo[2]=0;
foo[3]='c';
printf("%c %c %c %c %c \n",foo[0],foo[1],foo[2],foo[3]);
printf("%s\n",foo);
```

Use of sizeof() on a pointer type

Overview

Running sizeof() on a malloced pointer type will always return the wordsize/8.

Consequences

Authorization: This error can often cause one to allocate a buffer much smaller than what is needed and therefore other problems like a buffer overflow can be caused.

Exposure period

- Implementation: This is entirely an implementation flaw.

Platform

- Languages: C or C++
- Operating platforms: Any

Required resources

Any

Severity

High

Likelihood of exploit

High

Avoidance and mitigation

- Implementation: Unless one is trying to leverage running sizeof() on a pointer type to gain some platform independence or if one is mallocing a variable on the stack, this should not be done.

Discussion

One can in fact use the sizeof() of a pointer as useful information. An obvious case is to find out the word-size on a platform. More often than not, the appearance of sizeof(pointer)

Examples

In C/C++:

```
#include <stdiob.h>

int main(){
  void *foo;
  printf("%d\n",sizeof(foo)); //this will return wordsize/4
  return 0;
}
```

Related problems

Not available.

Unintentional pointer scaling

Overview

In C and C++, one may often accidentally refer to the wrong memory due to the semantics of when math operations are implicitly scaled.

Consequences

Often results in buffer overflow conditions.

Exposure period

- Design: Could choose a language with abstractions for memory access.
- Implementation: This problem generally is due to a programmer error.

Platform

C and C++.

Required resources

Any

Severity

High

Likelihood of exploit

Medium

Avoidance and mitigation

- Design: Use a platform with high-level memory abstractions.
- Implementation: Always use array indexing instead of direct pointer manipulation.
- Other: Use technologies for preventing buffer overflows.

Discussion

Programmers will often try to index from a pointer by adding a number of bytes, even though this is wrong, since C and C++ implicitly scale the operand by the size of the data type.

Examples

```
int *p = x;
char * second_char = (char *)(p + 1);
```

In this example, second_char is intended to point to the second byte of p. But, adding 1 to p actually adds sizeof(int) to p, giving a result that is incorrect (3 bytes off on 32-bit platforms).

If the resulting memory address is read, this could potentially be an information leak. If it is a write, it could be a security-critical write to unauthorized memory — whether or not it is a buffer overflow.

Note that the above code may also be wrong in other ways, particularly in a little endian environment.

Related problems

Not available.

Improper pointer subtraction

Overview

The subtraction of one pointer from another in order to determine size is dependant on the assumption that both pointers exist in the same memory chunk.

Consequences

- Authorization: There is the potential for arbitrary code execution with privileges of the vulnerable program.

Exposure period

- Pre-design through Build: The use of tools to prevent these errors should be used.
- Implementation: Many logic errors can lead to this condition. It can be exacerbated by lack of or misuse of mitigating technologies.

Platform

- Languages: C/C++/C#
- Operating Platforms: Any

Required resources

Any

Severity

High

Likelihood   of exploit

Medium

Avoidance and mitigation

- Pre-design through Build: Most static analysis programs should be able to catch these errors.
- Implementation: Save an index variable. This is the recommended solution. Rather than subtract pointers from one another, use an index variable of the same size as the pointers in question. Use this variable “walk” from one pointer to the other and calculate the difference. Always sanity check this number.

Related problems

Using the wrong operator

Overview

This is a common error given when an operator is used which does not make sense for the context appears.

Consequences

Unspecified.

Exposure period

- Pre-design through Build: The use of tools to detect this problem is recommended.
- Implementation: Many logic errors can lead to this condition. It can be exacerbated by lack, of or misuse, of mitigating technologies.

Platform

- Languages: Any
- Operating platforms: Any

Required resources

Any

Severity

Medium

Likelihood of exploit

Low

Avoidance and mitigation

- Pre-design through Build: Most static analysis programs should be able to catch these errors.
- Implementation: Save an index variable. This is the recommended solution. Rather than subtract pointers from one another, use an index variable of the same size as the pointers in question. Use this variable "walk" from one pointer to the other and calculate the difference. Always sanity check this number.

Discussion

These types of bugs generally are the result of a typo. Although most of them can easily be found when testing of the program, it is important that one correct these problems, since they almost certainly will break the code.

Examples

In C:

```
char foo;
foo=a+c;
```

Related problems

Not available.

Assigning instead of comparing

Overview

In many languages the compare statement is very close in appearance to the assignment statement and are often confused.

Consequences

Unspecified.

Exposure period

- Pre-design through Build: The use of tools to detect this problem is recommended.
- Implementation: Many logic errors can lead to this condition. It can be exacerbated by lack, or misuse, of mitigating technologies.

PlatforM

- Languages: C, C++
- Operating platforms: Any

Required resources

Any

Severity

High

Likelihood of exploit

Low

Avoidance and mitigation

- Pre-design: Through Build: Many IDEs and static analysis products will detect this problem.
- Implementation: Place constants on the left. If one attempts to assign a constant with a variable, the compiler will of course produce an error.

Discussion

This bug is generally as a result of a typo and usually should cause obvious problems with program execution. If the comparison is in an *if* statement, the *if* statement will always return the value of the right-hand side variable.

Examples

```
void called(int foo){
        if (foo=1)  printf("foo\n");
}
int main(){

        called(2);
        return 0;
}
```

Related problems

Not available.

Comparing instead of assigning

Overview

In many languages, the compare statement is very close in appearance to the assignment statement; they are often confused.

Consequences

Unspecified.

Exposure period

- Pre-design through Build: The use of tools to detect this problem is recommended.
- Implementation: Many logic errors can lead to this condition. It can be exacerbated by lack, or misuse, of mitigating technologies.

Platform

- Languages: C, C++, Java
- Operating platforms: Any

Required resources

Any

Severity

High

Likelihood of exploit

Low

Avoidance and mitigation

- Pre-design: Through Build: Many IDEs and static analysis products will detect this problem.

Discussion

This bug is mainly a typo and usually should cause obvious problems with program execution. The assignment will not always take place.

Examples

In C/C++/Java:

```
void called(int foo){
        foo==1;
        if (foo==1) printf("foo\n");
}
int main(){

        called(2);
        return 0;
}
```

Related problems

Not available.

Incorrect block delimitation

Overview

In some languages, forgetting to explicitly delimit a block can result in a logic error that can, in turn, have security implications.

Consequences

This is a general logic error — with all the potential consequences that this entails.

Exposure period

- Implementation

Platform

C, C++, C#, Java

Required resources

Any

Severity

Varies

Likelihood of exploit

Low

Avoidance and mitigation

Implementation: Always use explicit block delimitation and use static-analysis technologies to enforce this practice.

Discussion

In many languages, braces are optional for blocks, and — in a case where braces are omitted — it is possible to insert a logic error where a statement is thought to be in a block but is not. This is a common and well known reliability error.

Examples

In this example, when the condition is true, the intention may be that both *x* and *y* run.

```
if (condition==true) x;
  y;
```

Related problems

Not available.

Omitted break statement

Overview

Omitting a break statement so that one may fall through is often indistinguishable from an error, and therefore should not be used.

Consequences

Unspecified.

Exposure period

- Pre-design through Build: The use of tools to detect this problem is recommended.
- Implementation: Many logic errors can lead to this condition. It can be exacerbated by lack of or misuse of mitigating technologies

Platform

- Languages: C/C++/Java
- Operating platforms: Any

Required resources

Any

Severity

High

Likelihood of exploit

Medium

Avoidance and mitigation

- Pre-design through Build: Most static analysis programs should be able to catch these errors.
- Implementation: The functionality of omitting a break statement could be clarified with an if statement. This method is much safer.

Discussion

While most languages with similar constructs automatically run only a single branch, C and C++ are different. This has bitten many programmers, and can lead to critical code executing in situations where it should not.

Examples

Java:

```
{
    int month = 8;
        switch (month) {
            case 1:  print("January");
```

```
                    case 2:  print("February");
                    case 3:  print("March");
                    case 4:  print("April");
                    case 5:  println("May");
                    case 6:  print("June");
                    case 7:  print("July");
                    case 8:  print("August");
                    case 9:  print("September");
                    case 10: print("October");
                    case 11: print("November");
                    case 12: print("December");
                }
                println(" is a great month");
        }
```

C/C++:

Is identical if one replaces print with printf or cout.

Now one might think that if they just tested case12, it will display that the respective month “is a great month.” However, if one tested November, one notice that it would display “November December is a great month.”

Related problems

Not available.

Improper cleanup on thrown exception

Overview

Causing a change in flow, due to an exception, can often leave the code in a bad state.

Consequences

- Implementation: The code could be left in a bad state.

Exposure period

- Implementation: Many logic errors can lead to this condition.

Platform

- Languages: Java, C, C# or any language which can throw an exception.
- Operating platforms: Any

Required resources

Any

Severity

Medium

Likelihood of exploit

Medium

Avoidance and mitigation

- Implementation: If one breaks from a loop or function by throwing an exception, make sure that cleanup happens or that you should exit the program. Use throwing exceptions sparsely.

Discussion

Often, when functions or loops become complicated, some level of cleanup in the beginning to the end is needed. Often, since exceptions can disturb the flow of the code, one can leave a code block in a bad state.

Examples

In C++/Java:

```
public class foo {
  public static final void main( String args[] ) {
        boolean returnValue;
        returnValue=doStuff();
  }
  public static final boolean doStuff( ) {
        boolean threadLock;
        boolean truthvalue=true;

        try {
```

```
                        while(//check some condition){
                                threadLock=true;
                                //do some stuff to truthvalue
                                threadLock=false;
                        }
                } catch (Exception e){
                        System.err.println("You did something bad");
                                if (something) return truthvalue;
                }
                return  truthvalue;
          }
        }
```

In this case, you may leave a thread locked accidentally.

Related problems

Not available.

Uncaught exception

Overview

When an exception is thrown and not caught, the process has given up an opportunity to decide if a given failure or event is worth a change in execution.

Consequences

Undefined.

Exposure period

- Requirements specification: The choice could be made to use a language that is resistant to this issues.
- Implementation: Many logic errors can lead to this condition. It can be exacerbated by lack, or misuse, of mitigating technologies. Generally this problem is either caused by using a foreign API or an API which the programmer is not familiar with.

Platform

- Languages: Java, C++, C#, or any language which has exceptions.
- Operating platforms: Any

Required resources

Any

Severity

Medium

Likelihood of exploit

Medium

Avoidance and mitigation

- Requirements Specification: The choice between a language which has named or unnamed exceptions needs to be done. While unnamed exceptions exacerbate the chance of not properly dealing with an exception, named exceptions suffer from the up call version of the weak base class problem.
- Requirements Specification: A language can be used which requires, at compile time, to catch all serious exceptions. However, one must make sure to use the most current version of the API as new exceptions could be added.
- Implementation: Catch all relevant exceptions. This is the recommended solution. Ensure that all exceptions are handled in such a way that you can be sure of the state of your system at any given moment.

Examples

In C++:

```
#include <iostream.h>
#include <new>
#include <stdlib.h>

int
main(){
        char            input[100];
        int             i, n;
        long           *l;

Required resources        cout <<   many numbers do you want to type in? ";
        cin.getline(input, 100);
        i = atoi(input);
        //here we are purposly not checking to see if this call to
        //new works
        //try {
                l = new long    [i];
        //}

        //catch (bad_alloc & ba) {
        //       cout << "Exception:" << endl;
        //}
        if (l == NULL)
                exit(1);
        for (n = 0; n < i; n++) {
                cout << "Enter number: ";
                cin.getline(input, 100);
                l[n] = atol(input);
        }
        cout << "You have entered: ";
        for (n = 0; n < i; n++)
                cout << l[n] << ", ";
        delete[] l;
        return 0;
}
```

In this example, since we do not check if *new* throws an exception, we can find strange failures if large values are entered.

Related problems

Not available.

Improper error handling

Overview

Sometimes an error is detected, and bad or no action is taken.

Consequences

Undefined.

Exposure period

Implementation: This is generally a logical flaw or a typo introduced completely at implementation time.

Platform

Languages: All

Operating platforms: All

Required resources

Any

Severity

Medium

Likelihood of exploit

Medium

Avoidance and mitigation

Implementation: Properly handle each exception. This is the recommended solution. Ensure that all exceptions are handled in such a way that you can be sure of the state of your system at any given moment.

Discussion

If a function returns an error, it is important to either fix the problem and try again, alert the user that an error has happened and let the program continue, or alert the user and close and cleanup the program.

Examples

In C:

```
foo=malloc(sizeof(char);
//the next line checks to see if malloc failed
if (foo==0) {
//We do nothing so we just ignore the error.
}
```

In C++ and Java:

```
while (DoSomething()) {
    try {
      /* perform main loop here */
    }
    catch (Exception &e){
      /* do nothing, but catch so it’ll compile... */
    }
  }
```

Related problems

Not available.

Improper temp file opening

Overview

Tempfile creation should be done in a safe way. To be safe, the temp file function should open up the temp file with appropriate access control. The temp file function should also retain this quality, while being resistant to race conditions.

Consequences

- Confidentiality: If the temporary file can be read, by the attacker, sensitive information may be in that file which could be revealed.
- Authorization: If that file can be written to by the attacker, the file might be moved into a place to which the attacker does not have access. This will allow the attacker to gain selective resource access-control privileges.

Exposure period

- Requirements specification: The choice could be made to use a language or library that is not susceptible to these issues.
- Implementation: If one must use there own tempfile implementation than many logic errors can lead to this condition.

Platform

- Languages: All
- Operating platforms: This problem exists mainly on older operating systems and should be fixed in newer versions.

Required resources

Any

Severity

High

Likelihood of exploit

High

Avoidance and mitigation

- Requirements specification: Many contemporary languages have functions which properly handle this condition. Older C temp file functions are especially susceptible.
- Implementation: Ensure that you use proper file permissions. This can be achieved by using a safe temp file function. Temporary files should be writable and readable only by the process which own the file.
- Implementation: Randomize temporary file names. This can also be achieved by using a safe tempfile function. This will ensure that temporary files will not be created in predictable places.

Discussion

Depending on the data stored in the temporary file, there is the potential for an attacker to gain an additional input vector which is trusted as non-malicious. It may be possible to make arbitrary changes to data structures, user information, or even process ownership.

Examples

In C\C++:

```
FILE *stream;
char tempstring[] = "String to be written";

if( (stream = tmpfile()) == NULL ) {
   perror("Could not open new temporary file\n");
   return (-1);
}
/* write data to tmp file */
/* ... */
_rmtmp();
```

The temp file created in the above code is always readable and writable by all users.

In Java:

```
try {
    File temp = File.createTempFile("pattern", ".suffix");
    temp.deleteOnExit();
    BufferedWriter out = new BufferedWriter(new FileWriter(temp));
    out.write("aString");
    out.close(); }
catch (IOException e) { }
```

This temp file is readable by all users.

Related problems

Not available.

Guessed or visible temporary file

Overview

On some operating systems, the fact that the temp file exists may be apparent to any user.

Consequences

Confidentiality: Since the file is visible and the application which is using the temp file could be known, the attacker has gained information about what the user is doing at that time.

Exposure period

- Requirements specification: The choice could be made to use a language or library that is not susceptible to these issues.
- Implementation: If one must use his own temp file implementation, many logic errors can lead to this condition.

Platform

- Languages: All languages which support file input and output.
- Operating platforms: This problem exists mainly on older operating systems and cygwin.

Required resources

Any

Severity

Low

Likelihood of exploit

Low

Avoidance and mitigation

- Requirements specification: Many contemporary languages have functions which properly handle this condition. Older C temp file functions are especially susceptible.
- Implementation: Try to store sensitive tempfiles in a directory which is not world readable — i.e., per user temp files.
- Implementation: Avoid using vulnerable temp file functions.

Discussion

Since the file is visible, the application which is using the temp file could be known. If one has access to list the processes on the system, the attacker has gained information about what the user is doing at that time. By correlating this with the applications the user is running, an attacker could potentially discover what a user's actions are. From this, higher levels of security could be breached.

Examples

In C\C++:

```
FILE *stream;
char tempstring[] = "String to be written";

if( (stream = tmpfile()) == NULL ) {
   perror("Could not open new temporary file\n");
   return (-1);
}
/* write data to tmp file */
/* ... */
_rmtmp();
In cygwin and some older unixes one can ls /tmp and see that this temp file exists.
```

In Java:

```
try {
    File temp = File.createTempFile("pattern", ".suffix");
    temp.deleteOnExit();
    BufferedWriter out = new BufferedWriter(new FileWriter(temp));
    out.write("aString");
    out.close(); }
catch (IOException e) { }
```

This temp file is readable by all users.

Related problems

Not available.

Failure to deallocate data

Overview

If memory is allocated and not freed the process could continue to consume more and more memory and eventually crash.

Consequences

- Availability: If an attacker can find the memory leak, an attacker may be able to cause the application to leak quickly and therefore cause the application to crash.

Exposure period

- Requirements specification: The choice could be made to use a language that is not susceptible to these issues.
- Implementation: Many logic errors can lead to this condition. It can be exacerbated by lack of or misuse of mitigating technologies.

Platform

- Languages: C, C++, Fortran, Assembly
- Operating platforms: All, although partial preventative measures may be deployed depending on environment.

Required resources

Any

Severity

Medium

Likelihood of exploit

Medium

Avoidance and mitigation

- Pre-design: Use a language or compiler that performs automatic bounds checking.
- Design: Use an abstraction library to abstract away risky APIs. Not a complete solution.
- Pre-design through Build: The Boehm-Demers-Weiser Garbage Collector or valgrind can be used to detect leaks in code. This is not a complete solution as it is not 100% effective.

Discussion

If a memory leak exists within a program, the longer a program runs, the more it encounters the leak scenario and the larger its memory footprint will become. An attacker could potentially discover that the leak locally or remotely can cause the leak condition rapidly so that the program crashes.

Examples

In C:

```
bar connection(){
  foo = malloc(1024);
  return foo;
}
endConnection(bar foo){
  free(foo);
}
int main() {
  while(1)
    //thread 1
    //On a connection
    foo=connection();

    //thread 2
    //When the connection ends
    endConnection(foo)
  }
}
```

Here the problem is that every time a connection is made, more memory is allocated. So if one just opened up more and more connections, eventually the machine would run out of memory.

Related problems

Not available.

Non-cryptographic PRNG

Overview

The use of Non-cryptographic Pseudo-Random Number Generators (PRNGs) as a source for security can be very dangerous, since they are predictable.

Consequences

- Authentication: Potentially a weak source of random numbers could weaken the encryption method used for authentication of users. In this case, a password could potentially be discovered.

Exposure period

- Design through Implementation: It is important to realize that if one is utilizing randomness for important security, one should use the best random numbers available.

Platform

- Languages: All languages.
- Operating platforms: All platforms.

Required resources

Any

Severity

High

Likelihood of exploit

Medium

Avoidance and mitigation

- Design through Implementation: Use functions or hardware which use a hardware-based random number generation for all crypto. This is the recommended solution. Use CyptGenRandom on Windows, or hw_rand() on Linux.

Discussion

Often a pseudo-random number generator (PRNG) is not designed for cryptography. Sometimes a mediocre source of randomness is sufficient or preferable for algorithms which use random numbers. Weak generators generally take less processing power and/or do not use the precious, finite, entropy sources on a system.

Examples

In C\C++:

```
srand(time())
int randNum = rand();
```

In Java:

```
        Random r = new Random()
```

For a given seed, these “random number” generators will produce a reliable stream of numbers. Therefore, if an attacker knows the seed or can guess it easily, he will be able to reliably guess your random numbers.

Related problems

Not available.

Failure to check whether privileges were dropped successfully

Overview

If one changes security privileges, one should ensure that the change was successful.

Consequences

- Authorization: If privileges are not dropped, neither are access rights of the user. Often these rights can be prevented from being dropped.
- Authentication: If privileges are not dropped, in some cases the system may record actions as the user which is being impersonated rather than the impersonator.

Exposure period

- Implementation: Properly check all return values.

Platform

- Language: C, C++, Java, or any language which can make system calls or has its own privilege system.
- Operating platforms: UNIX, Windows NT, Windows 2000, Windows XP, or any platform which has access control or authentication.

Required resources

A process with changed privileges.

Severity

Very High

Likelihood   of exploit

Medium

Avoidance and mitigation

- Implementation: In Windows make sure that the process token has the SeImpersonatePrivi- lege(Microsoft Server 2003).
- Implementation: Always check all of your return values.

Discussion

In Microsoft operating environments that have access control, impersonation is used so that access checks can be performed on a client identity by a server with higher privileges. By impersonating the client, the server is restricted to client-level security — although in different threads it may have much higher privileges.

Code which relies on this for security must ensure that the impersonation succeeded — i.e., that a proper privilege demotion happened.

Examples

In C/C++

```
bool DoSecureStuff(HANDLE hPipe){ {
   bool fDataWritten = false;
   ImpersonateNamedPipeClient(hPipe);
   HANDLE hFile = CreateFile(...);
   /../ RevertToSelf()/../
}
```

Since we did not check the return value of ImpersonateNamedPipeClient, we do not know if the call succeeded.

Related problems

Not available.