Vulnerabilities in C/C++ programs – Part II

TDDC90 – Software Security

Ulf Kargén

Department of Computer and Information Science (IDA)

Division for Database and Information Techniques (ADIT)

Integer overflows and sign errors

Adding, subtracting, or multiplying an integer with a too large value can cause it to wrap-around

- Can be used to circumvent input validation to e.g. cause buffer overflows

```
void print_user(char* username) {
   char buffer[1024];
   char* prefix = “User: “;
   const unsigned int prefix_len = 6;

   unsigned int len = strlen(username);

   // Space required for prefix, username and
   // string terminator.
   unsigned int size = prefix_len + len + 1;
   if(size > 1024)
       exit_with_error();  // Error, too long string

   strcpy(buffer, prefix);   // Copy prefix
   strcat(buffer, username); // Concatenate username

   printf(“%s”, buffer);
}
```

What happens if the user supplies an extremely long ‘username’ here?

- If username is longer than UINT_MAX - 7, an integer overflow will occur.

⇨ The length check will succeed, but more than 4GB copied into buffer…

Integer overflows and sign errors

A similar class of vulnerabilities are sign errors – mixing signed and unsigned data types in an unsafe way

```
// Reads ‘size’ bytes from file ‘f’ into buffer ‘out’
void
read_from_file(void* out, FILE* f, unsigned int size);
...

int read_entry(FILE* input)
{
   char buffer[1024];
   int len;

   // Read four-byte length field from file into ‘len’
   read_from_file(&len, input, 4);

   if(len > 1024)
       return ERR_CODE; // Error, too long string

   // Read ‘len’ bytes from file into buffer
   read_from_file(buffer, input, len);
   ...
```

The problem here is that signed and unsigned data types are mixed.

- What happens if the length field in the file is a negative number, e.g. -1?

⇨ The length check will succeed, as -1 < 1024

⇨ In the call to ‘read_from_file’, the ‘len’ variable will be interpreted as an unsigned data type

⇨ The 32-bit representation of -1 is 0xFFFFFFFF ≈ 4 billion, way more than the buffer size!

Integer overflows and sign errors

Can be extremely subtle!

If the length check from previous example is changed from this…

```
if(len > 1024)
   return ERR_CODE; // Error, too long string
```

… to this, the code is no longer vulnerable. Why?

```
if(len > sizeof(buffer))
   return ERR_CODE; // Error, too long string
```

- The value returned by the ‘sizeof’ operator is always of an unsigned type (size_t)
- According to the C standard, if two values of different data types are compared, and one of the types can represent larger numbers than the other, the value of the smaller type is implicitly cast to the larger.
- The above comparison becomes `if((size_t)len > sizeof(buffer))`
- … but don’t rely on these sort of things to avoid vulnerabilities :-)

Avoiding integer errors

- Again: Perform input validation!
  - Catch e.g. negative lengths of strings, etc.
- Avoid mixing signed and unsigned data types, as well as types of different sizes. **Heed compiler warnings!**
- Understand sizes and conversion rules for data types!
- Use the type ‘size_t’ for variables representing lengths of things. ‘size_t’ is always an unsigned data type (cannot be negative).
- Check for wraparounds :

```
size_t A = ...
size_t B = ...
if(A > SIZE_MAX - B)
   error(); // Overflow
size_t sum = A + B;
...
```

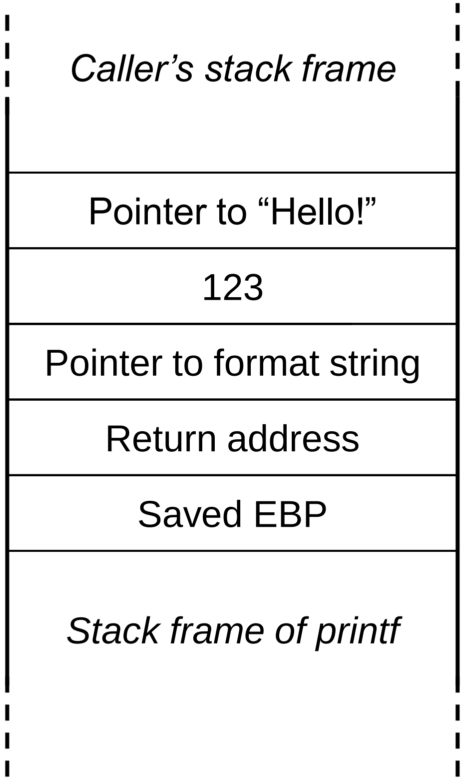
Format string bugs

The printf-family of functions are used in C to format output.

- Takes a format string with placeholders for variable output fields, and a number of arguments corresponding to placeholders in string.

```
printf("An integer: %d, a string: %s", 123, "Hello!");
// Output: An integer: 123, a string: Hello!
```

- Vulnerability stems from lazy programmers writing `printf(string_from_user)` instead of `printf("%s", string_from_user)`
  - This works fine, as long as the user-controlled string doesn't contain format specifiers!
- printf simply assumes that arguments corresponding to all format specifiers exist on the stack – will output whatever is on the stack if that is not the case!
- Supply e.g. a string "%X%X%X%X" to output four 32-bit words from callers stack frame in hexadecimal notation – trivial information disclosure.
  - Also possible to read memory at arbitrary address with some trickery.



Format string bugs

- printf also has little known (and used) format specifier %n that is used to store the number of written characters so far into a variable

```
printf("A string: %s%n", "Hello World!", &x);
// Output: A string: Hello World!
// x == 22 after execution
```

- Can be used by attacker to write arbitrary data to arbitrary address in memory!
- Idea (to write arbitrary 32-bit value):
  - Supply the address to write to *in the format string itself*
  - Use a (large) number of format specifiers to advance printf's internal argument pointer to *the format string in the caller's stack frame* (to get to the write address)
  - Control value written by controlling length of string
  - Repeat four times, writing one byte at a time
- Details not important here – available in extra reading material for interested students.

Avoiding format string bugs

- Use `printf("%s", str)` instead of `printf(str)`
  - Unless, perhaps, str is a (hardcoded) constant string
- Format string bugs can fairly easily be spotted with static analysis (use of non-constant string as first argument)
- Modern compilers usually warn about (some) insecure use of printf-family of functions.

Non-memory-corruption vulnerabilities

So far, we have looked at bugs allowing attackers to overwrite control-data for arbitrary code execution or DoS

- Many dangerous types of bugs are not the result of buffer overflows or other memory corruption errors:
  - Race conditions
  - Out-of-bounds reads of data

Race conditions

A shared resource is changed between check and use

check_validity_of_user_data()

[…]

use_user_data()

- Example: File system race conditions

```
if (access(filename, W_OK) == 0) {
   if ((fd = open(filename, O_WRONLY)) == NULL) {
       perror(filename);
       return -1;
   }

   /* Write to the file */
}
```

- What if file changes between access-check and open?
- Attacker can e.g. replace real file with symbolic link with same name to sensitive file (e.g. /etc/passwd on Unix)

Avoiding race conditions

- Very broad class of vulnerabilities
  - Race conditions on file system
  - Race conditions on memory access between threads
  - etc.
- See literature on course web page for recommendations on avoiding file race conditions in Unix

Out-of-bounds reads
Case study: Heartbleed

Out-of-bounds read from heap-allocated memory in OpenSSL allows attackers to read out certificates, private keys, sensitive documents, etc…

- Due to incorrect implementation of *heartbeat* extension of TLS
- One of the parties in a connection can send a payload with arbitrary data to the other party, which echoes it back unchanged to confirm that it is up and running.
- Problem: Length of payload that is echoed back is not checked. Can read past actual payload into adjacent memory!

Out-of-bounds reads

Case study: Heartbleed

```
int
dtls1_process_heartbeat(SSL *s)
    {
    unsigned char *p = &s->s3->rrec.data[0], *pl;
    unsigned short hbtype;
    unsigned int payload;
    unsigned int padding = 16; /* Use minimum padding */
    ...
    /* Read type and payload length first */
    hbtype = *p++;
    n2s(p, payload);
    pl = p;
    ...
```

‘p’ points to data in SSL record

Record consists of:
Heartbeat type (1 byte)
Payload length (2 bytes)
Payload data (up to 65536 bytes)

‘pl’ points to payload data

Copy length of payload into ‘payload’

LiU EXPANDING REALITY

Out-of-bounds reads

Case study: Heartbleed

```
...
unsigned char *buffer, *bp;
int r;

/* Allocate memory for the response, size is 1 byte
 * message type, plus 2 bytes payload length, plus
 * payload, plus padding
 */
buffer = OPENSSL_malloc(1 + 2 + payload + padding);
bp = buffer;
...
/* Enter response type, length and copy payload */
*bp++ = TLS1_HB_RESPONSE;
s2n(payload, bp);
memcpy(bp, pl, payload);
```

Allocate heap memory for reply

Copy ‘payload’ bytes into buffer for reply message

Problem: The length of ‘payload’ is never checked! Sender can claim a payload length longer than the actual received SSL record.

- ⇨ Up to 64 kB of adjacent heap memory can be leaked to attacker.
- ⇨ Has been shown to allow reading out private keys from servers!

Writing secure code

Secure coding practices and principles

- Principles to adhere to
  - Best practices
  - Secure coding standards

- Patterns of code to use or to avoid
  - Secure architectural and design patterns
    - ⇨ Covered in lecture on Secure software development

- Library functions to use or to avoid

CERT top 10 Secure Coding Practices

1. Validate input
2. Heed compiler warnings
3. Architect and design for security policies
4. Keep it simple
5. Default deny
6. Adhere to the principle of least privilege
7. Sanitize data sent to other systems
8. Practice defense in depth
9. Use effective quality assurance techniques
10. Adopt a secure coding standard

CERT C Secure Coding Standard (excerpt)

Recommendations

- INT01-C: Use rsize_t or size_t for integer values representing size of an object
- MSC15-C: Do not depend on undefined behavior
- SRC06-C: Do not assume that strtok() leaves the parse string unchanged
- FIO07-C: Prefer fseek() to rewind()
- MEM01-C: Store a new value in pointers immediately after free()

Rules

- INT32-C: Ensure that operations on signed integers to not result in overflow
- MSC33-C: Do not pass invalid data to the asctime() function
- STR33-C: Size wide character strings correctly
- FIO31-C: Do not open a file that is already open
- MEM32-C: Detect and handle memory allocation errors

SDL Banned Function Calls

CharToOem, CharToOemA, CharToOemBuffA, CharToOemBuffW, CharToOemW, IsBadCodePtr, IsBadHugeReadPtr, IsBadHugeWritePtr, IsBadReadPtr, IsBadStringPtr, IsBadWritePtr, Makepath, OemToChar, OemToCharA, OemToCharW, StrCat, StrCatA, StrCatBuff, StrCatBuffA, StrCatBuffW, StrCatChainW, StrCatN, StrCatNA, StrCatNW, StrCatW, StrCpy, StrCpyA, StrCpyN, StrCpyNA, StrCpyNW, StrCpyW, StrLen, StrNCat, StrNCatA, StrNCatW, StrNCpy, StrNCpyA, StrNCpyW, _alloca, _fstrncat, _fstrncpy, _getts, _gettws, _i64toa, _i64tow, _itoa, _itow, _makepath, _mbccat, _mbccpy, _mbscat, _mbscpy, _mbslen, _mbsnbcat, _mbsnbcpy, _mbsncat, _mbsncpy, _mbstok, _mbstrlen, _snprintf, _sntprintf, _sntscanf, _snwprintf, _splitpath, _stprintf, _stscanf, _tccat, _tccpy, _tcscat, _tcscpy, _tcsncat, _tcsncpy, _tcstok, _tmakepath, _tscanf, _tsplitpath, _ui64toa, _ui64tot, _ui64tow, _ultoa, _ultot, _ultow, _vsnprintf, _vsntprintf, _vsnwprintf, _vstprintf, _wmakepath, _wsplitpath, alloca, gets, lstrcat, lstrcatA, lstrcatW, lstrcatn, lstrcatnA, lstrcatnW, lstrcpy, lstrcpyA, lstrcpyW, lstrcpyn, lstrcpynA, lstrcpynW, lstrlen, lstrncat, nsprintf, scanf, snscanf, snwscanf, sprintf, sprintfA, sprintfW, sscanf, strcat, strcatA, strcatW, strcpy, strcpyA, strcpyW, strcpynA, strlen, strncat, strncpy, strtok, swprintf, swscanf, vsprintf, vswprintf, wcscat, wcscpy, wcslen, wcsncat, wcsncpy, wcstok, wnsprintf, wnsprintfA, wnsprintfW, wscanf, wsprintf, wsprintfA, wsprintfW, wvnsprintf, wvnsprintfA, wvnsprintfW, wvsprintf, wvsprintfA, wvsprintfW

Mitigations

OS and compiler exploit protections

Exploit mitigations

Mitigations are technical measures meant to make attacks *harder*

- Raises cost (time required, expertise) for attackers
- But doesn’t necessarily make all attacks impossible

Implemented in either operating system or compiler

- Stack cookies (Compiler based)
- DEP (OS based)
- ASLR (OS based)

Stack cookies

- Implemented in compiler, must be applied during compilation
- A stack *cookie* or *canary* is inserted in stack frame before the return pointer
- Cookie is checked prior to executing ‘ret’ instruction. If it has changed, program is terminated with an error message.
  - ⇨ Impossible for attacker to overwrite return pointer with a *buffer overflow* without altering cookie.

- Typical implementation works approximately like this:
  - Cookie placed before saved EBP – prevents overwrite of both return address and saved EBP
  - Cookie stored in global variable that is randomly generated at program startup
    - ➢ Static cookies won’t work, can just be replicated by attacker!
  - A call to a function that checks cookie integrity is inserted before ‘ret’ instruction. Terminates program if cookie doesn’t match original.
  - Typically also reorders local variables in stack frame so that buffers (arrays) are located first – prevents overwrites of e.g. function pointers in local variables.

Stack cookies

Example

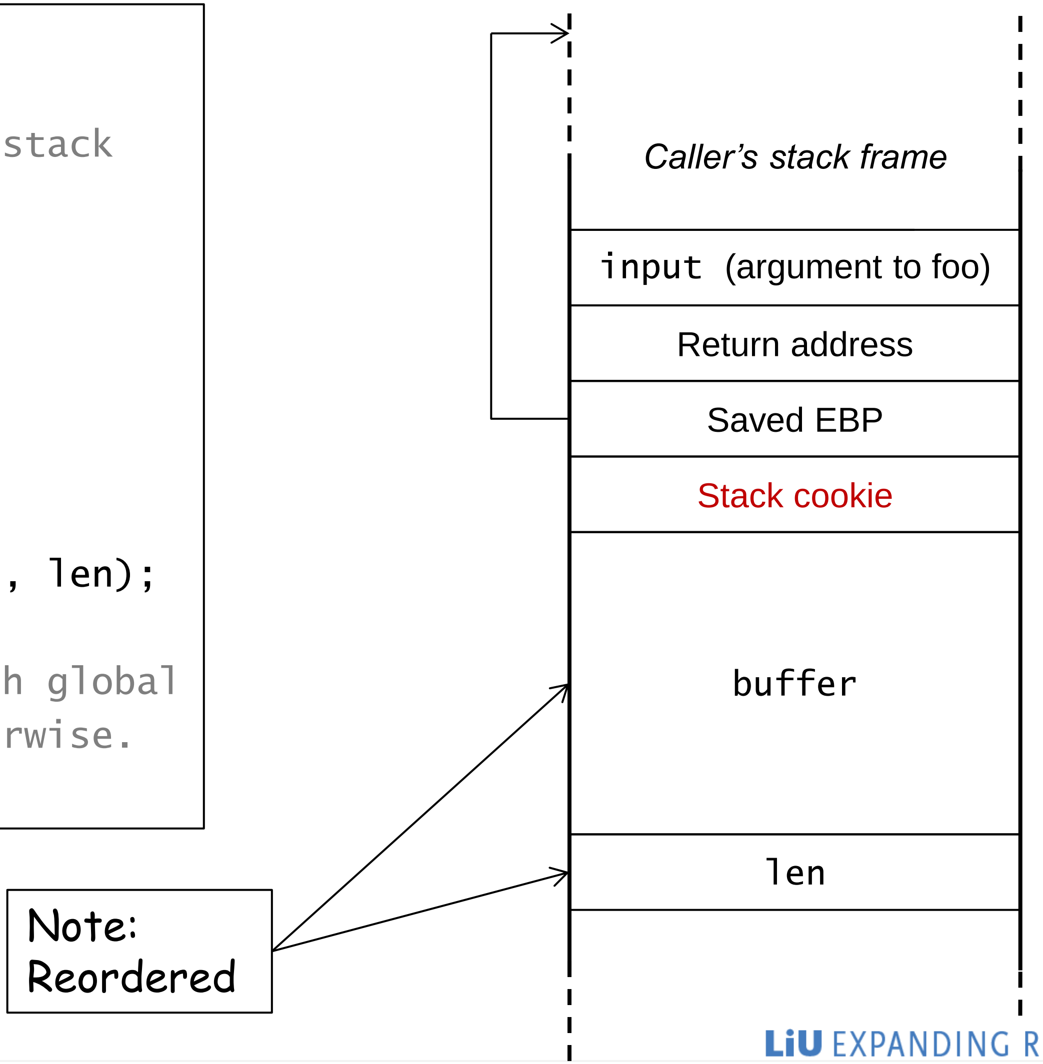
```
void foo(char* input)
{
   // Push global cookie to stack

   unsigned int len;
   char buffer[16];

   len = strlen(input);
   strcpy(buffer, input);

   printf(“%s: %d\n”, buffer, len);

   // Check that cookie match global
   // cookie. Terminate otherwise.
}
```



Defeating stack cookies

- **Only** mitigates stack-based buffer overflows
- Applying stack cookies comes at a cost – for small functions that are called frequently, cost of cookie check can be significant
  - ⇨ Not applied to all functions – various heuristics to determine where to use stack cookies
  - ⇨ Only used in functions with buffers of certain types and sizes – some attacks may still be possible
- On Windows, the Structured Exception Handler (SEH) record on the stack can be overwritten to take control before the return and cookie check

Data Execution Prevention

Use hardware-enforced nonexecutable data pages to prevent shellcode from running

Implemented in many different operating systems under different names

- OpenBSD: W^X (Write xor Execute)
- Windows: Data Execution Prevention (DEP)
- Linux: Variants of the PaX MPROTECT patch for Linux kernel

Data Execution Prevention

Recall: Virtual memory divided into pages (typically 4 kB on x86)

- Pages can be marked as Readable, Writable, and Executable
  - ⇨ Write to non-Writable page results in program termination (Segmentation fault)
- Older CPUs (prior to ~2005) didn’t have hardware support to enforce the Executable permission
  - ⇨ Possible to execute code from pages marked as non-Executable
- Modern CPUs have this – the NX-bit (for No eXecute)
  - ⇨ Setting all pages for stack, heap, etc. as non-Executable prevents shellcode from executing.
  - ⇨ Effectively mitigates all code execution exploits from previous slides.

Defeating DEP
The return-to-libc attack

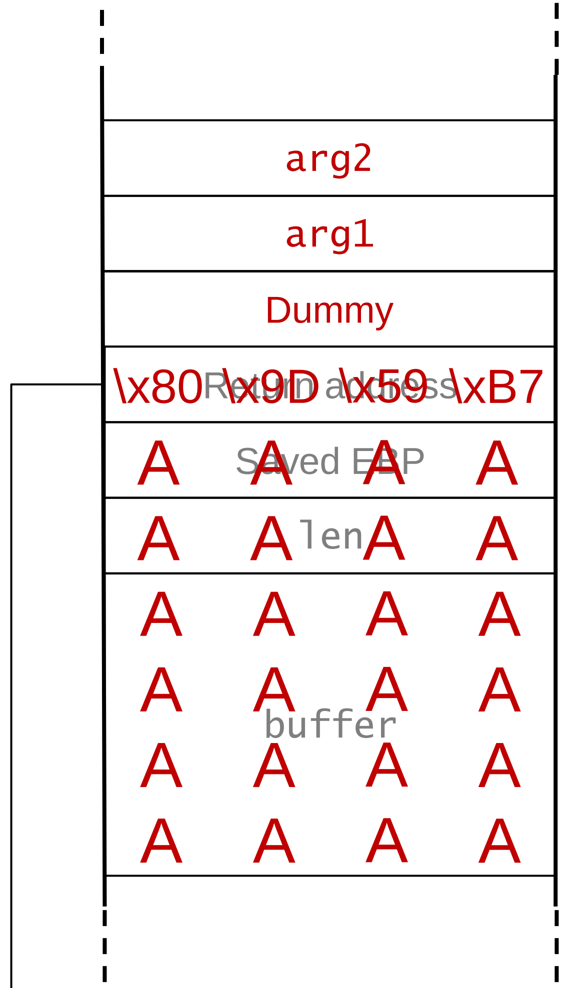
Instead of injecting executable code, *re-use existing function* within program

- Overflow stack buffer to set up stack to look like a function call is about to be made
- Overwrite return pointer to “return” into start of desired function
  - ⇨ No code on the stack is executed – DEP won’t help
- Functions within the standard C library (libc) are popular targets, since libc is present in address space of (almost) every program. Hence the name.
  - E.g. the ‘system’ library function is popular – executes an arbitrary shell command with privileges of calling program

return-to-libc example

Recall the stdcall calling convention:

- Caller pushes arguments from right to left to stack.
- The ‘call’ instruction pushes return address to stack and jumps to first instruction of called function
- To “call” function `bar(int arg1, int arg2)` using return-to-libc:
  - Overwrite return pointer with address to first instruction of ‘bar’
  - Put a dummy value above return pointer. This is where ‘bar’ expects the caller’s ‘call’ instruction to have put the return address.
  - Put the arguments to ‘bar’ in correct order on the stack.
  - At ‘ret’ instruction, ‘bar’ will be “called”, and ESP will point at the dummy “return address”, just like in a real call.




```
; Start of ‘bar’
push ebp
mov ebp, esp
...
```

LiU EXPANDING REALITY

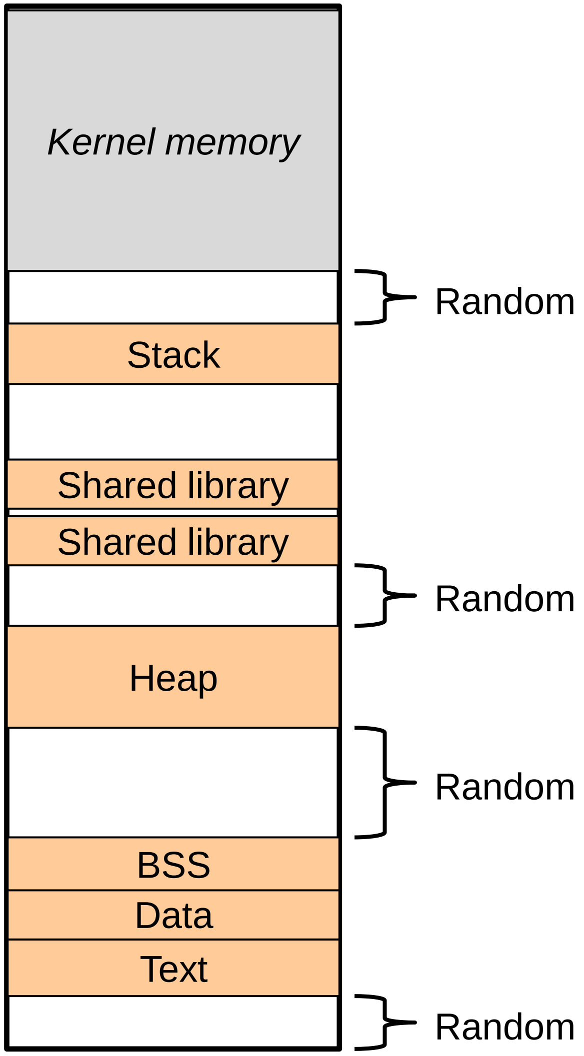
return-to-libc limitations

- Limited to using existing functions within program address space
- Calling functions which takes pointers (e.g. strings) as arguments is tricky.
- Cannot perform calls where one argument is required to have the value zero (Why?)

Address Space Layout Randomization (ASLR)

Observation: Most exploit methods rely on predicting the address of some piece of code or control data.

- Idea: *Randomize* position of heap, stack, main executable, shared libraries, etc. to prevent attacks.
  - New positions each time program is started
- Very effective at mitigating many kinds of attacks.
- Brute forcing still possible on 32-bit machines, where the memory space available for randomization is small. (Works mostly for local exploits.)
- Methods that do not rely on predicting addresses are still effective
  - The *relative* position of data within the same segment is unaffected by ASLR
  - Still possible to e.g. overwrite sensitive non-control data on stack or heap



”Modern” exploit methods

A brief overview

Heap Spraying

Defeats: ASLR

- Applicable in certain scenarios where user controllable input can exert large control over heap allocations
- Make the program allocate large numbers of large memory blocks, filling most of the heap.
  - Each block consists of a large NOP sled followed by shellcode.
- When hijacking control flow of program, e.g. through a stack based-buffer overflow, jump to random position in the middle of the heap

  ⇨ Large probability of hitting one of the NOP sleds.
- Typically requires a scriptable environment. Popular when e.g. attacking web browsers
  - Create large arrays with e.g. JavaScript, and fill them with NOPs + shellcode.

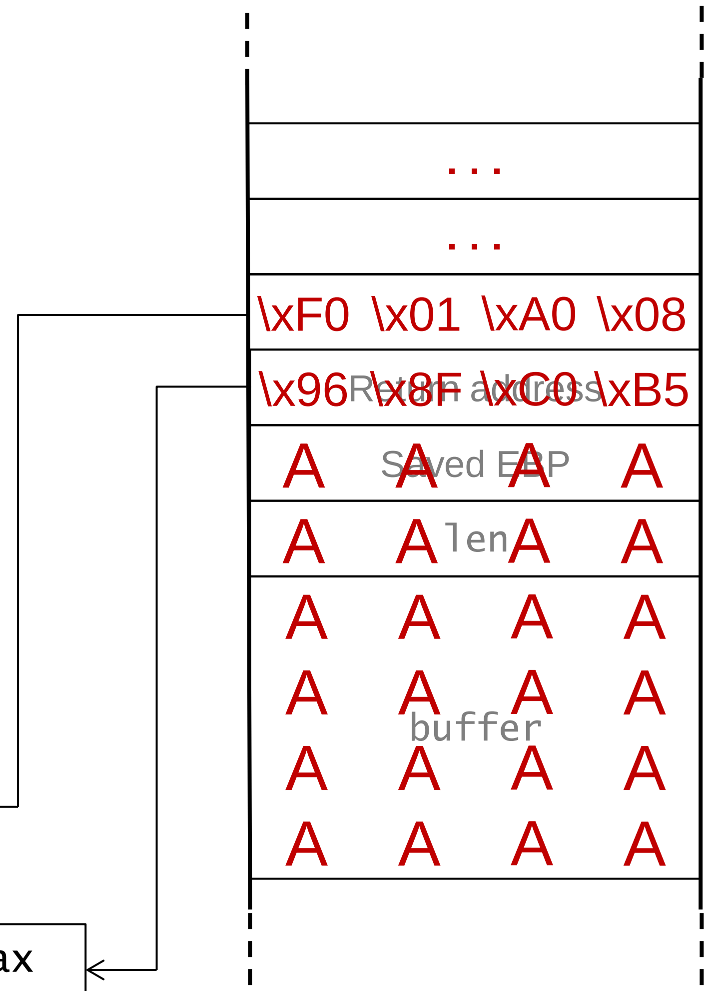
Return Oriented Programming (ROP)

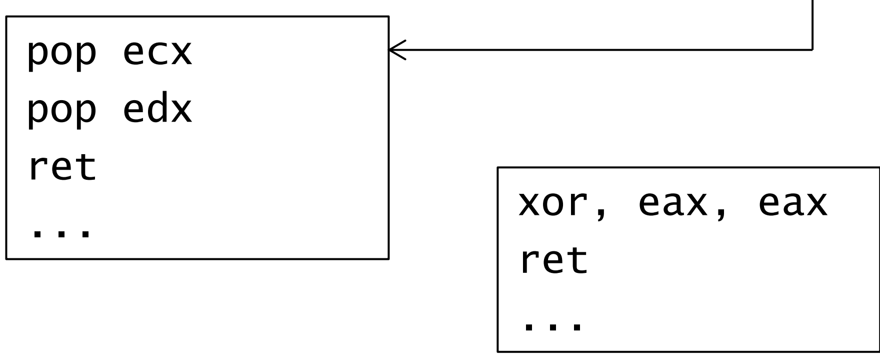
Defeats: DEP

- The “standard” method used today by attackers to bypass DEP
- Generalization of return-to-libc
- First proposed by Hovav Shacham in 2007
  - Showed that a Turing complete “language” could be created by re-using code of an executable.
- Allows *arbitrary* code execution without injecting any code – completely circumvents DEP!
- Idea: Identify code snippets of the form
  ```
  [do something useful]
  ret
  ```
  in existing code (main executable or libraries).
  - Such snippets are referred to as *gadgets*

Return Oriented Programming (ROP)

- Put addresses of gadgets on the stack, the first one overwriting the return pointer.
  - This “chain” of addresses is often referred to as a ROP chain.
- When the executing function returns, it will pop the gadget address, jump to the gadget, execute the useful instruction(s), and then “return” to the next gadget, and so on.
- Shacham showed that even complex program constructs, such as loops, can be constructed in this way.





Return Oriented Programming (ROP)

ROP attacks rely on being able to predict the addresses of gadgets, and are thus mitigated by ASLR – given that the positions of *all* executable memory regions are randomized.

- Often not the case in practice
  - On Linux, the executable file itself is usually not randomized, while shared libraries are.
  - On Windows prior to Windows 8, the default is that all executables need to “opt in” with a special flag set at compile time to be randomized.
    - Many legacy libraries are still not compiled with this flag, and are potential targets for a ROP attack.

Effectiveness of mitigations

- No mitigation is a silver bullet
- Some attack methods are thwarted, but often still possible to craft exploits
- However, standard techniques often don't work "out of the box"
  - Often need to combine many different attack techniques, several different vulnerabilities, and program or OS-specific "tricks"
- Example:

1. Take advantage of a flaw in particular ASLR implementation, or find target-specific non-randomized executable memory regions to create ROP chain.
2. Set of gadgets typically limited in practice, create small ROP payload that disables DEP, and jumps to traditional shellcode.
3. Possibly utilize heap spraying or information leakage bugs to locate shellcode in memory

Effectiveness of mitigations

- Bottom line: Crafting exploits still possible, but requires considerable expertise and time.
  - ⇨ People rarely write exploits “for fun” anymore
- Instead:
  - Professional penetration testers
  - Organized crime
  - Intelligence agencies

- A previously unknown vulnerability (“zero-day”) in popular software with reliable exploit can be worth several $100 000…