TDDC03 Projects, spring 2004

Applied Security Auditing

A practical study on analysis guidelines for written code

Magnus Holmgren, Wolfgang Mähr
(magho442|wolma000)@student.liu.se

Linköpings universitet
May 10, 2004

Project no. 010

Supervisor: John Wilander

Applied Security Auditing

Magnus Holmgren
magho442@student.liu.se

Wolfgang Mähr
wolma000@student.liu.se

Abstract

As security auditing and security testing always rely on the experience of the involved persons, here we develop a process that should be easier to apply and that should need less experience. This process contains six methods, each resulting in useful criteria for testing and auditing each. These methods are explained and then demonstrated on a small Java web server and some examples are explained further on an open source web server written in Pike. Furthermore, evaluations of these methods with advantages and disadvantages are provided. In addition, an example of the resulting report is provided.

1. Introduction

1.1. What Is Security, Why Do We Need It and How Do We Estimate It?

Security has grown increasingly important in the field of computer science the last years. The security of a system says how vulnerable the system is to intentional attacks. The vulnerability is usually measured in three dimensions called *CIA*: *Confidentiality*, *Integrity* and *Availability*. Confidentiality means that information cannot be accessed by unauthorized users. Integrity means that valid information cannot be changed (by accident or maliciously), at least not without leaving a trace, and also that the origin of the data is what is expected. Finally, availability means that a service cannot be limited or shutdown by any other instance.

These confidentiality and availability demands contradict each other. With the ubiquitous availability of the Internet, we want to take advantage thereof. We want to do billing, reservations, etc. over the Internet, we want to consult our doctor via Internet instead of waiting for ages in the waiting room. With our demand of ubiquitous accessibility, we are also opening our data to everybody with access to the Internet and with the knowledge of our access information. Through this way, a malicious person can try to access private data from almost everywhere, but still this access information cannot be too complicated for users (the appropriate users) to remember and to use.

The field of computer science is not so young as it seems. There is already now a huge amount of existing software and working solutions available and often it is required that some existing code is extended, integrated or just depended on. For these cases, it is important to know how secure and reliable this older piece of software is. It may often occur that such an estimation is hard to do because meaningful documentation is missing (source code, product information, etc).

Still, some methodologies should be applicable to be able to see possible threats and to take in account problems resulting from this integration or usage. Right now, there are three main methods for this task: *analysis tools*, *security audits* and *security testing*. Analysis tools are programs that go through the code and try to find security leaks (unsafe function calls, security estimations for code, etc). Security auditing is the process to go through code with an auditing team and trying to figure out how secure a piece of code is. The third method is security testing, to run the code and look for security leaks. All of these methods are possible to be done on compiled code, but knowledge about the interior or access to the code is an advantage for these methods.

1.2. The Goal of This Study

This study is the result of a project on software security for java web servers. Our goal of the project was to analyze a web server with the help of different methods (*Threat Scheme*, *Attack Tree*, and some other methods, all are described later) and to report our experiences applying these methods. With this report however, we try to shift away from just reporting the security issues of the software we got to work on, towards describing the general process of doing such a security analysis. We will use the web servers we analysed as example to demonstrate the whole process of security auditing software. This process is the main result of this study. In both cases, we can do the whole process, which sometimes may not be possible, because code or some documentation is missing. Furthermore, we had to adapt various methods to be applicable in the way we wanted to use them. These methods, explained later on, are usually used in development instead of analysis but their task is quite similar: finding possible security leaks. Therefore, we try to describe the whole process and evaluate each method.

Our motivation for this guide is also that until now there are these three possibilities to check if some code is secure, but the audit and testing rely a lot on experience and knowledge. We now wanted to develop a more methodical approach to find test cases and criteria for evaluating the security of software. With this methodology, we hope to provide a complete result because it is achieved by proceeding systematically and which therefore is easier to reproduce. In addition, we hope that these test cases and these criteria are found faster and easier than in the less organized way. In addition, we hope that these test cases and these criteria are found faster and easier than in the less organized way.

As the basis of our work, we will use two books: "Building Secure Software" by John Viega and Gary McGraw [1] and "Writing Secure Code" by Howard [2].

1.3. Scenario

We assume for this analysis that we have a finished product and try to evaluate the security of it. This product should be used for internal use or should be integrated into an application that is being developed. Another possible scenario is that we have to re-evaluate an old piece of our software to find problems in those lines of code.

The thing that all scenarios have in common is that neither we nor anybody else changes the evaluated code during the evaluation. This means that even if we have access to the code, the methods are all still applied to the same code. This code may then be changed later on as a result of this audit. Even if we just have the documentation and no code, we want this process to be applicable. Therefore, it is possible that we do not use any analysis tools but that we rely on manual testing. The developed methodology should then help to create specific test cases and other criteria to achieve the results.

1.4. The Tested Software

We will analyze one small web-server written in Java (called Dahlberg) and a bigger one written in Pike (called Caudium). The Dahlberg we will use as an example for the whole process, while we will use Caudium for just pointing out specific issues. Unless otherwise stated, we always refer to Dahlberg. To understand these two web servers, some knowledge of their basic concepts is needed. We gathered this information from different places in the web, namely the Java security whitepaper [4] and the SecurityFocus advisories [5]:

1.4.1. Java

The Java platform, announced in 1995 by Sun Microsystems and Netscape, is more than a programming language. Java provides a platform-independent *runtime environment* and *API*. Java source code is not compiled into machine code, but instead translated into a semi-compiled form called *bytecode*. The bytecode is then interpreted and executed by the Java Virtual Machine or *JVM*. JVM's exist for a large number of platforms and provide the machine-independent environment. A set of standard classes provide an interface to hardware and operating system services, as well as useful functionality usually not available across all platforms. There are no pointers, only object variables, which can contain either a reference to an existing object or `null`. This means that buffer overflows and stray pointers causing overwritten memory are impossible. Memory allocation and deallocation is handled automatically by the garbage collection. The typing of the language is strong. Since all bytecode is interpreted by the JVM, controlling what the code can do or cannot do is relatively easy. This results in the Java sandbox, a well-confined program space with restricted access to the computer resources. Sandboxing is most useful with mobile code (i.e. applets, automatically downloaded and run when the user visits a web page) but it can also be used with stand-alone applications like for example the Dahlberg web server. Even though Java is a very controlled language (lack of pointers, strong typing of language, etc.), bugs can sneak into the code. This can happen in the program code as well as in the JVM implementations. The latter make all Java software vulnerable. Most vulnerabilities reported in the Java Virtual Machine are relevant primarily in the mobile code situation. For example, a early vulnerability in Netscape Navigator 2.0 concerned the rule that an applet should only be allowed to communicate with the server from which it came, but the JVM did not check the IP address, but only the host name, allowing an attach. Other points of failure are the *class loader*, which is responsible for fetching the proper class files when needed, and the Bugs in those areas could render not only hosts running applets vulnerable, but also affect server software.

1.4.2. Dahlberg

The Dahlberg web server is a very simple web server written in Java by Per Dahlberg. It only supports the GET method to fetch static content. It does not support any CGI, SSI or other means of creating dynamic output. Its most advanced feature is that it can generate directory listings. The server handles requests by creating handler threads processing the requests that are accepted by the

main thread and the passed forward. The configuration is very simple and straightforward.

1.4.3. Pike

According to the Pike website [6], it is "[...] a dynamic programming language with a syntax similar to Java and C. It is simple to learn, does not require long compilation passes and has powerful built-in data types allowing simple and really fast data manipulation." What was to become Pike was a hobby project at the beginning, which later was then used by Roxen Internet Software (formerly known as InfomationsVävarna) for their web server. Nowadays, Pike is maintained by IDA, the department of Computer and Information Science at Linköping University. Pike is a more C-like language than Java. Also, like Python or Java, it is interpreted, not compiled. Pike programs are translated into an intermediate form and then run, but unlike Java, this translation is done on every invocation. Therefore, Pike is useful for scripting. It also uses a garbage collector to relieve the programmer from having to keep track of used memory. The data types in Pike are the basic types int, float and string. In addition, like in most languages except C and C++, strings are safe. There is no way to read or write past the end of a string; there is no way to reference an invalid memory location at all. A slight disadvantage of Pike is that there is no language construct like the `synchronized` keyword in Java, which is used in Java to make critical parts thread safe. A resultless search on the web for security issues in Pike, could have several reasons: Either there are no known vulnerabilities in Pike, or no one has found Pike important enough to report any vulnerabilities. This leads to the assumption that using Pike is not more insecure than using any other language.

1.4.4. Caudium

Caudium 1.2.x is a full-fledged web server based on an earlier version of Roxen Challenger by Roxen IS and available under the GPL License. It is written in Pike (except for a core library written in C), which also makes Pike the natural choice for programming dynamic content. Still support for other languages, such as PHP, can be added with help of a separate module. RXML, the Roxen Macro Language, is another way to create simple server-side scripts by writing special tags between the HTML code. Modules provide various kinds of functionality and new modules can be written or installed by the site administrator to change the behaviour of Caudium via web interface.

2. Security Analysis

2.1. The Process of Analysis

We will take the Dahlberg Web server as the example to work on, whereas Caudium will be used for showing some specific issues. We will take a top-down approach to the code, which means that we will start looking at the application from an abstract black-box view with its interaction with the environment and then we will step deeper into the application and the code in each iteration. In each iteration, we will first describe the applied method theoretically, then practically apply it on our example(s) and finally comment on the method in means of our personal experiences. We will also make personal recommendations and mention ideas to these methods. The reason for using this kind of layout for our audit is that the process can be quit after each iteration. It should be possible after each iteration to use fresh insights to do testing and auditing on the software. Hopefully the results improve by each iteration, because each iteration removes one layer of abstraction. It should be possible to quit the process in the middle of two methods and still get acceptable results and criteria. As some methods need code review, which cannot be applied on a software package where the source code is not accessible, those methods have to be skipped sometimes. Then, after applying these methods to collect criteria for testing and auditing, they can take place. Another advantage of this process is that issues covered earlier can be omitted later because they do not have to be covered twice. Therefore the process should be fast. The final result should be a security report saying how secure the product code is and which problems are hidden in the code; we will end up with such a report for the example application.

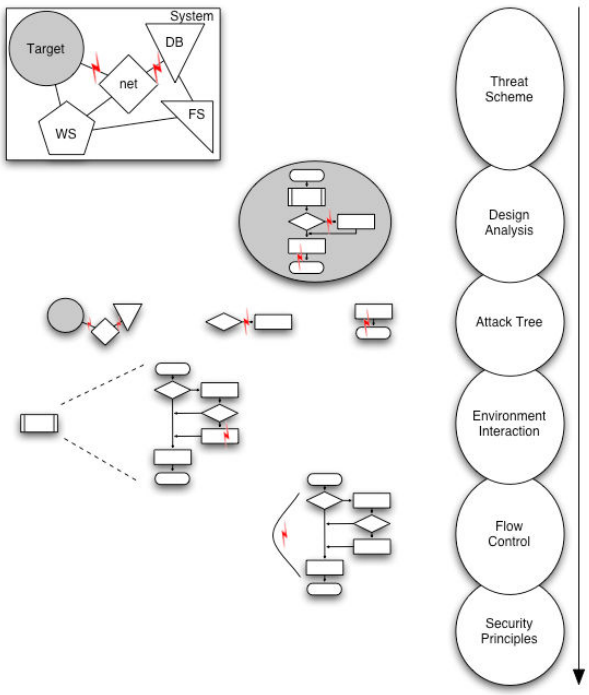

Figure 1. Auditing process

2.2. Threat Scheme

2.2.1. The Theory Behind

This method is based on threat models described in the book of Howard [2]. Threat models are used in the design phase. According to Howard they are the most effective thing to do for design. To begin with, a threat model gives the developer a first deeper contact with the whole system and a first glimpse on the solution. Moreover, this method forces the developer to think about security in relation to this problem already very early and therefore a lot problems are be found early.

Threat Schemes should do the same as threat models, but for security analysis. Threat Schemes should help finding some first security issues, but their main intention is to give the reviewers an approach to the targeted software. They should force the reviewers to think about the environment in which the software is used and with which it is interacting. For this Threat Scheme we just take the first part of the threat models and focus on the environment without looking into the application. The closer look on the interior of this blackbox will then come in the next step (Design Analysis).

For this first approach, we take a black-box view of the program and model the program and its interaction with its environment. This should be possible by looking at the documentation of the product. Maybe some things have to be tested (i.e. used resources) if the documentation is bad, but still this step should be applicable in any case. In our example, we have to model how the interaction works and try to think of possible problems that could occur in each interaction. We will not decide if these problems can occur in this specific application, but instead we just collect as many ideas as possible. This also means that internal problems, like input validation, control flow and failing modes, will not be covered here, only the possible environmental failures. In the Second step we can then describe the problems found in the analysis of the model. Here we should try to be as precise as possible. This description helps us defining a possible check in step three. This check can demand either a "code review", "security testing" or an "Attack Tree". In any way, it also makes sense to add as much information as possible for the check here, so that this can be done easier later on.

2.2.2. Applying the Method

Understanding of complex issues is easier by models and diagrams and we have to think about them, when we are creating them. Therefore, we first make a general diagram of the environment of the software (*fig.2*). In case such or a similar diagram is already present, we also can use that one.

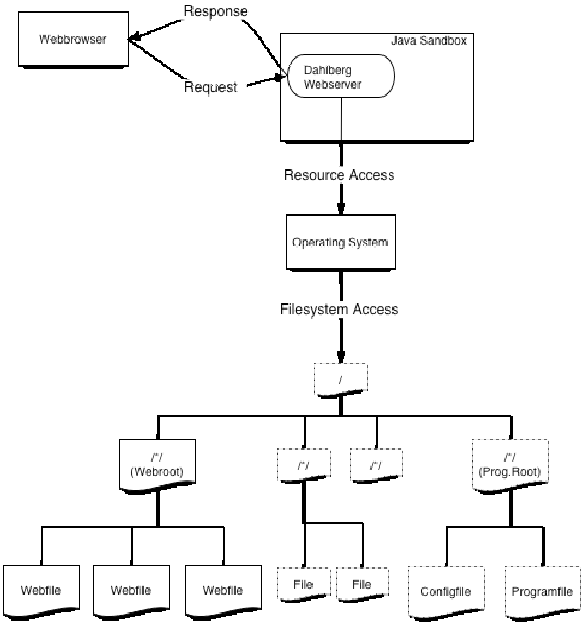

Figure 2. Threat diagram of the web server and its environment

From this diagram, we can read all interactions between the software and the environment (the arrows) that have to be tested. In the next step we can extract a list of possible problems there (*table 1* in the appendix).

With this table, we have found the first criteria for evaluation via testing and auditing. Some problems are marked with "Attack tree" as check which means that we have to cover these issues detailed in the Attack Tree, because they are too complex to be checked by a simple check or a code review and because they can be exploited in different ways. For example can file execution by an attacker be achieved by buffer overflow or tampering with the configuration. Anyway, these issues are beyond the scope here; they are inside of the black-box, and therefore we just note them and handle them later. The other simpler issues have been listed here, and for those we can give an estimate.

The last step of this test would then be to test and review the code for the criteria we found before. However, because this does not make sense right now – we do not have any specific knowledge about the code yet – we procrastinate these checks to the next phase.

2.2.3. Evaluation of the Method

We believe that this method gives a quite good first insight into the application and its environment. It also forces the auditor to get into first contact with the software and the environment. It is an important point that the environment is regarded as well when auditing the software, because security leaks can also depend on the environment. The exposure of security flaws usually happens through the environment. It is possible that a user can access the configuration files of the web server, but this is not possible over the network. Such leaks in the environment therefore still pose a threat to the application. Here the auditor is also introduced to the main functionality of the program, just like when designing the application. In our case, we have already found some security related questions about the software, which could be checked quite easily because the application is rather small. In bigger applications it is probably better to leave out the check and in the first step only focus on the collection of ideas about possible security threats. In the next step, we will then replace the black-box and start looking into the application.

One thing that should not be forgotten when applying this method is how much documentation of this application is available. In this phase it is easy to enter "Code review" as a possible check, but it might be, that the source code is not available. Also it might happen, that the source code is available, but cannot be tested for whatever reason (platform problems, etc). It might even happen, that only the design documentation is available (i.e. when different components are developed simultaneously by different companies and no real code is present) what makes it still harder to check for specific problems. However, at least possible problems can be found and mentioned in the report.

In case of "Code review" and "Security testing" we should also try to define how this method should be applied as exactly as possible. We should define what should be looked for in the review and how the test should look like. Furthermore, it should be considered that a code review in this phase might need long time, because of the high level of abstraction; the occurring problems may be big and complex and may span over many lines of code, which would then have to be reviewed. Also testing for these problems can be exhausting. In these cases, it makes sense to mark these problems to be checked by an Attack Tree, because they help checking systematically for complex problems. Further information on this can be found in the according chapter (Attack Trees).

To sum up, we can state that this first method makes sense to be spent time on, because we can already filter some issues here, so that the Attack Tree will not grow too big. This will save us time later on. The other, real advantage is that this method forces the auditor to focus on the environment. The drawback of this method is that it is abstract and therefore application specific issues cannot be found.

2.3. Design Analysis

2.3.1. The Theory Behind

In the next step, we move into the black-box and analyse the design of the target software. In this step, we rely on a design diagram. This diagram should ideally be available in the documentation. If they are not available, but we can access the code, we can try to reverse engineer such a diagram. This diagram may be a *flow diagram*, a *context diagram* with its use cases or a *domain model*. The domain model is for one reason not too good to have: It is static; the domain model just explains the relation between different classes. If there is another diagram additionally to the domain model, like a *collaboration diagram* or a *sequence diagram*, then the task of analysing the design is easier. Depending on the type of the diagram the threats are found at different positions. In a flow diagram we find the threats at the blocks, whereas in the in the context-diagram, the collaboration diagram and the sequence diagram we find them in the actions. In the domain model, these threats are also in the class interconnections, hidden in terms of just visible through the class

interface. Therefore, we should have one additional diagram of another type, if we are willing to use a domain model.

In this diagram, we try to figure out where problems might occur. If we have a diagram describing interactions (flow chart, context diagram, sequence diagram, etc.), our focus will be on these interactions when analysing, whereas if we have a domain model, we will focus on the relations of these classes. In the next step, we then can take the problems we found in the diagram and make a table with the threat name, its description the planned/possible behaviour and the possible check for that. The name should be descriptive so that we can distinguish the threats by their names. The description should explain the threat in more detail, so that everybody reading the analysis can understand what we talk about. For the planned/possible behaviour, we have to look at the diagram again and think of how the modelled piece of code should behave and how it could behave according to the diagram. This contains an estimation of the behaviour in the best and in the worst case. Finally, we also should think of a way to check how the program reacts in reality. This is again one of the three remarks: "Code review", "Security testing" or "Attack tree". As before, the Attack Tree should be used if the problem might be too complex to analyze here and as before, a description for the check is helping later on. We still have to keep in mind that this does not guarantee that all issues are covered, so for example race conditions or combinations of different problems can still produce unpredictable behaviour in more complex programs.

2.3.2. Applying the Method

In our example, we do not have any developer documentation; we will instead perform some analysis of the code to reverse-engineer a meaningful diagram. Since there are only a few classes in this application, we decide to make a flow diagram of the program. This diagram will not be too specific; we do not need to model each function call for this analysis. Instead, we just take blocks of code and describe them. This impreciseness in this phase is no problem, because we still just work on models. If we would model each function call, we could directly forgo this modelling and do a code review instead. Anyways, if we get a diagram from the documentation, this will also include some level of simplification and will not list all function calls. We also leave out error handling, because this will be covered later in the Flow Control Check; also because in a good design error handling should not include any program logics. The result of this analysis is the following diagram (Figure 3).

.

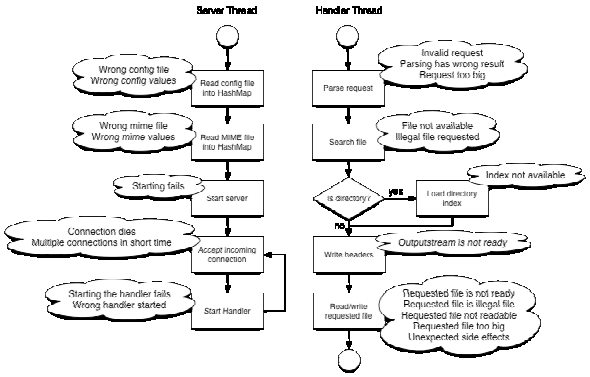

Figure 3. Design Diagram flowchart

In the diagram, we already integrated the next step. In that step we do the same as in the Threat Scheme; we search for possible threats to each block. For example, we have the threats that the wrong configuration file is read or that it contains the wrong values, as well as the searching for a file poses the threat that the searched file cannot be found. At the end of this step, we end up with a list of threats, of which we have to check each, whether it is real. Again, we make a table with the threats, their description, the planned or possible behaviour of the program and a possible check. These columns have exactly the same meanings as they had in the Threat Scheme. The name is for distinguishing the threats easily; the description helps to give the reader (and also the writer) a more specific idea about the threat. The planned/possible behaviour is meant to describe the best case and the worst case according to the diagram, if this threat occurs and the possible check is a first idea about how to check if this threat is real. One thing about the worst case: This is estimation is subjective. Depending if a shutdown of the server is seen worse than execution of arbitrary code, the entry in this column will be different. Actually, this is not the worst thing to happen, because the intention of this column is mainly to give an idea about how big the threat is, for a possible priority ranking and for the final report.

In Table 1 (found in the appendix) we now try to make this estimation.

Again, this table gives us some more specific information about possible vulnerabilities of the tested piece of software. As the last step of this method, we now have to check all the possible vulnerabilities found by this method to see if they are real threats. Because we left out this check in the method before, we also have to check the threats found there.

When we start to look at the code we can make the following conclusions (related issues from method one and two are now taken together):

Malformed request: Code just accepts GET method. Any relative address in the document request is tried to be accessed, character encoding is the issue of Java and the OS. Furthermore, basic requests just containing "GET /some/location" are executed even if not standard compliant.

Big requests: Everything is read but just the first line is parsed. Unless there is some limit on the length of lines read, very long lines could cause big memory allocations.

Request answered by the wrong process: No forwarding in the application, just the port assignment by the OS during startup. This means that the only forwarding is done by the OS and out of scope.

Responding to the wrong address: Tampering impossible, because the answer is automatically sent to the requesting address.

Too big response: Every file that is requested will be delivered, so the only place to limit the size of the answers is to keep the files shared small.

Response contains the wrong data: The response just contains standard HTTP signatures, MIME types and the contents of the requested files.

Changing system configurations/limitations: The code does not set any configurations, nor any Sandbox or OS setting. Furthermore, files are only read, but not written.

Overloading resources: There is an unlimited amount of threads answering the requests. If the response takes long time, several threads can be running at a time. Other requests are served. Ten threads usually should be on stand-by.

Wrong configuration values: Ability to change listening port, index and MIME files and document root. Influences confidentiality and availability. File access needed for exploit.

Wrong MIME file or wrong MIME values: Void MIME-type in HTTP response, which is an integrity issue. File access needed for exploit.

Starting fails: Server is not running.

Connection dies: Connection communication fails safely, the request is not proceeded.

Multiple connections in short time: Requests are served after each other, the server slows down.

Starting the handler fails: Request is not served.

Wrong handler started: Random code can be executed, threatening confidentiality, integrity and availability. Local file system access is needed for this because source code or files have to be changed

Output-stream is not ready: Connection fails safely, request is not served.

Invalid request: The request is parsed minimally, so errors in the request target are handed over to the operation system, the rest is neglected. No problem in the program functionality.

Parsing has wrong result: The parsing works properly for all needed cases.

File not available: 404 Error is returned - fails safely.

Illegal file requested: This is possible with directory traversal.

Index not available: Directory listing is returned - no problem here.

Requested file is not ready: Program waits until the file is readable. Response may be delayed

Requested file is illegal file: Fails safely - request not served.

Requested file not readable: Cannot occur. In the worst case the encrypted/raw text is sent back.

2.3.3. Evaluation of the Method

Depending on the size of the evaluated software, the size of this method may grow. Still, due to the abstraction of a diagram, we do not have to look at all lines of code, but on the other hand the amount of possible problems is bigger than in a code review. Because we do not focus on a specific implementation, we instead have to think about all the problems that could occur in an implementation.

Another problem is that the modelling might take a while if the software is big, because all code has to be gone through. In that case, it seems to be wiser just to review the code, instead of taking this step. The reason is that even though the looking at the code to generate a diagram may be faster than reviewing the code, creating a model may take more time, especially when more models have to be made. In this case it is in our opinion too much effort compared to directly reviewing the code. If documentation and models of the software are available, we think that it surely makes sense to use this method. One main reason is that this also gives the reviewer more insights on the design and its tidiness. Another advantage is that this method filters out most of the basic problems that can be found in program behaviour. For example, simple problems like "Connection dies" or "Invalid request" can be handled already here. Consequently, we do not have to check these possibilities in the Attack Tree later, which saves us time there. The danger in this method is that side effects and problems that are more complex cannot be found. In addition, it happens quite easily that a problem is estimated simpler than it is and is checked in this step, instead of constructing a more sophisticated Attack Tree. When we take for example the possible threat of "Wrong configuration values", on the first sight it seems that there cannot happen too much. It might be, like in this example, that the server listens on the wrong port or that the document root

is at the wrong place, but if the configuration file has more power and decides which classes are loaded and where they can be found, then a possible change of the configuration may pose a big risk. Consequently, we have to be really careful in this method to not take the too simplistic solution on the question, if something is a threat. Instead, we have to keep in mind that things could be more complex as they look on the diagram we are working on.

As said before, it also depends on the diagram that can be used. In a domain model it is rather hard to find any possible threats, because just the interfaces of all domain objects are visible, but not their interaction. Thus, from such a diagram it is possible to find problems like "ClassA not loaded in time" or "Network not available" but time and sequence related issues are hard to find because the diagram does not contain any time information. In addition, more basic problems, such as "illegal file requested", are rather hard to find because the action of a file being requested is hidden behind a method declaration that might look in *UML* like

```
+ read_file(String filename): String
```

which stands for a function called `read_file` that takes a filename and returns a string and is publicly visible. Such an interface does not provide too much information what happens behind the facade (in the basic intention of modelling it also should not do that).

2.4. Attack Tree Analysis

2.4.1. The Theory Behind

Another way to check if a piece of software is secure is the use of checklists. As there usually are some known issues for each type of application (i.e. directory traversal, cross-site scripting or SQL-injection for web servers), it also makes sense to create a checklist of these issues and check whether the software is vulnerable to those.

An Attack Tree - described in the book of Viega [1] - is a way to structure and get an overview over the attacks that are or might be possible against a system or a piece of software. At the top level are the goals an attacker might have when attacking. On the next level are the possible attacks to achieve the goal and further down the attacks are decomposed into possible and/or necessary steps to carry out the attack. This top-down approach can help to analyze which attacks can be done on a specific piece of software. In the end we have a list of problems that can occur and all their possible extensions. Furthermore, the Attack Tree includes possible requirements for such an attack to be successful.

2.4.2. Applying the Method

In this example of such an Attack Tree, we start with such a list of known issues. Then we go further and start breaking down the problem into smaller sub problems and combination of problems. Describe the problem roughly (e.g. "Illegal File Access") and then break it down into smaller problems by splitting them. This splitting can be made either by distinguishing between different types (e.g. "Write access" vs. "Read Access") or between different ways to attack (e.g. "Accessing via the program itself" vs. "Accessing via random executed code"). The whole and more general Attack Tree can be found in the appendix, whereas we here just focus on the issues found in the Dahlberg web server. This listing does not deal with script engines, database managers and other components that the web server might incorporate or use. This is mainly because they are not used in the web server yet, but also because that would be out of scope of this analysis. Those should still be analyzed when they are incorporated into the web server.

Issues we found through the Attack Tree:

Unauthorized access to information: Simply request any desired file by directory traversal with and even without special characters. This is also possible by creating symbolic links on the local (server) file system. With local access, it is also possible to tamper with the configuration file and its values.

Execute code as the web server account: Tampering with the configuration makes this possible as already described before

Access and/or modify other users' information handled by the server: This can be done by the directory traversal, because the server is running as root, so everything is readable.

Intercept other users' communication with the server: Intercept unencrypted traffic, which is possible, since the web server only supports HTTP, which is an insecure protocol.

Denial-of-service attack: This can be done either by flooding the server with requests or by requesting too big files. Naturally, it is also possible to bring the server down with local file access on the server in the program directory. Additionally, both Caudium and Dahlberg have a problem with memory consumption that could allow a DoS attack: They accept infinitely long request strings and infinitely many header lines, even malformed ones. With the following command, Caudium will quickly begin allocating all available memory:

```
(echo GET / HTTP/1.0; yes) | nc hostname port
```

(nc is netcat, the TCP/IP Swiss army knife.)

Dahlberg is not vulnerable to exactly the attack mentioned above, as it throws away all headers (it does not use them), but it may run into trouble with extremely long lines. It depends on how the readLine() method of the BufferedReader class works.

Some comments on a number of attacks/vulnerabilities: A *directory traversal* attack can be possible when the check for a requested file interprets the request differently than the code that really fetches it. As an example we could have a validation function disallowing double dot ("..", meaning "parent directory") in the file name, so that the requested file lies within the boundaries of the server root (the directory on the local system that corresponds to http://server.name/). If this function is used before the URL is decrypted. It is possible to change to parent directories with encrypting the dot with "%2E" which is translated during the decryption into just this dot. The Dahlberg web server does not check at all what file is requested. Thus, this attack is possible here. Caudium, on the other hand, relies on a built-in function in Pike that translates a directory specification containing "." and/or ".." components to an equivalent one without such components. If, in the process, a ".." would lead above the root directory, it is simply ignored.

Privilege escalation happens when a legitimate local or a user is able to gain the privileges of a more privileged user. If the server will be serving files owned by an untrusted user, much care has to be taken to protect the system from malicious code that could be executed by the server. Therefore, the server should, if it itself runs under a privileged account, be able to handle certain or all requests under an unprivileged account. The reason for this is that if scripts are run by the server as root, a malicious user could just run own malicious code as root, when it is requested. With this code, it is then possible to access any other file. With symbolic links, the range of files that the web server is allowed to access may be enlarged. Another way of abusing a program to run with higher privileges is to mess with the configuration. The program should therefore make sure that the configuration file(s) it reads is properly protected, and otherwise refuse to start, or at least warn the administrator. Caudium lets ordinary users write scripts that the server will execute. Therefore, It is important that these scripts cannot access files that the owner of the files cannot access. Dahlberg never runs any external code, so this case does not apply. Still, it might be possible to mess with the configuration.

Denial-of-service attacks for web servers can have various forms. We will not explain all of them, but shortly show some variations. Firstly, if the server can handle only a limited number of parallel requests (either due to system resources or configured limits), then each request should not be allowed to take more than a limited amount of time to fulfil. Otherwise, requests can be blocked later. Secondly, all kinds of bugs, like unhandled division-by-zero exceptions or memory leaks, could crash the server. Therefore, proper error handling is vital for the server. Even if an error cannot be handled gracefully, it should not make the whole server crash.

Even if it is the system administrator's responsibility of to make sure that files can only be accessed by the right persons, it is easy to oversee something and thereby accidentally create a loophole that could compromise the whole system. Therefore, default settings and file permissions should always rather too restrictive than too loose. Even if this limits the availability of a system, it still ensures that an attacker cannot easily read files that should not be readable or use a default password to access a system.

2.4.3. Evaluation of the Method

An Attack Tree is a good way of organizing and structuring information about a subject of security analysis. Due to its tree shape, it is helping to focus first on the abstract level and then go into detail. The structure of the tree also helps to find new branches by looking at analogies and the basic version of taxonomy helps as well. This means that if we have a branch for "denial-of-service attack" as a child of the "crash the server branch" it seemed for us to be easy to think then about how to crash the server without using a DoS. Still, there is some creativity needed to find security flaws, especially when it is about finding new flaws and not starting from the list of known issues. This also shows that there is no standard algorithm or any scientific method that says how to build an Attack Tree.

After completing the tree, it seems to make sense to rate the problems by severity and ease of taking advantage over these. In our example, we started but then decided to skip this part because all severe issues found were already found earlier in the process, so we did not need to do any deeper analysis on the problems. In general, however, we planned to rate these threats by giving the severity as well as the ease of taking advantage a rating from one to five. For the ease of taking advantage, we also used the value zero, which stands for "not applicable on this application". Multiplying these two numbers then tells us how big the threat is. In other words: An easy to exploit leak that can be seriously harmful is a big problem.

One thing that we experienced when using this Attack Tree was that it really helps to make complex problems easier, since it helps breaking down big problems in

smaller parts. On the other hand, we can also see that except the "interception of user data" we already found all issues in the methods before. Still, we found more details and more different possibilities of exploiting the problems using the Attack Tree. In the case of "unauthorized access to information", for example, we found directory traversal vulnerabilities. However, we just thought of the traversal with the double dots to roam in the file system, and we did not consider any other way to code the path until we had the branch in the tree. The reason is simple: Since we first considered what threats there are, we were satisfied with finding threats for example like "Illegal file requested". In that method we did not think of how that threat can happen. However, in the Attack Tree we do, because we were thinking if there is any other way to access the files somewhere else, without using the double dots. This is in our opinion the main advantage of the Attack Tree. It makes the reviewer think of the problem in another way.

2.5. Environment Interaction Check

2.5.1. The Theory Behind

As a piece of software interacts with its environment (user, sensors or other software), there is always the possibility that the communication contains void data or information. Therefore, it is extremely important to check how the software handles received data. There are well-known examples of input validation problems in combination with buffer overflow, for example in O'Reilly's "Input Validation in C and C++" [3].

This check is the first code review; therefore, we go through the code to check how the received input is handled. This is the first of the review methods and we decided that the most important criteria for reviewing the security of program code are its validation of input and its standards compliancy of output. By input and output we refer to the interaction with the environment, not the in- and output in means of reading from and writing to the memory. Here, all input has to be checked; the user input via direct interaction (GUI or system.in) and the input via communication with other software (network communication, configuration files). It is important to check how and if the input is validated. For this task, we look at the validation methods and check if they really just validate good input – if the good input is modelled for the validation. There are two main reasons for just modelling good input, of which the first one is "the less features, the more secure". This means that there are fewer possibilities to have problems with the validation. The second reason is that it is not always possible to model all bad input, because there might be infinite variations of bad input and some could easily be overlooked. If there are cases where we are not totally sure about the behaviour, we have to test with pitfalls. The problem with testing is that we cannot test everything; we can just test a finite number of cases. Therefore, it is ideal if we can formally prove the input check valid.

The method we use here is to go through the code and look for environment interaction (input and output). We make a table with all interactions and write down the file name, the line numbers, the type of action, the check result and the severity. The file name and the line numbers are to find it again and for the reader's orientation. The type of interaction should help to give an idea about what the code does, without looking at the code. The check result is a brief summary about what the code validates and what is right or wrong about the way the code validates. The severity rating helps then, at the end of the method, to give a ranking of how severe a threat is. This ranking is made subjectively and is of no major importance but it should help to get the right order into the found issues when the report has to be written. We suggest a scale with 5+1 values, meaning a scale with the values "very low", "low", "medium", "high", "very high", and finally the value "none", if there is no risk for problems at all.

If the result of the check is not totally clear it might help to write a test for the specific code and check it.

2.5.2. Applying the Method

In our example, we have source code in four files, three of which do input and output. Two of those are reading configuration files and HandleRequest.java is doing the main user communication, handling and answering requests. The two other files have some problems validating the input well, because they handle configuration files. The problem with configurations is that they do not necessarily have specific values, but sometimes it is possible to check for syntactical correctness. This was not done in the configuration loading classes, probably because it is too much effort. By having a configuration file written XML or as a Java .configuration file, the intention is to have a simple file type where all values can easily be read into a data structure (hash map or similar) and then easily be accessed in the program. If there is a plausibility check added into the reading, then this simplicity is lost.

Here is the summary of the issues we have found (the main table is *table 3* in the appendix):

- MIMETypes.java: **Read configuration:** The input cannot be validated because almost anything can be valid input. Void input does not harm the application but sets some wrong parameters in the response. Prop-

erties without values are possible. These problems are rated "very low".

- ConfFile.java: **Read configuration:** The input can not be validated because almost anything can be valid input. Void input can harm the application in making it crash or not find the needed data. Properties without value are not possible. No check for a valid configuration. These problems are considered as "low".
- HandleRequest.java: **Parse request:** Basic parsing, not all of the request is parsed or validated. The URI is not validated. We consider this issue as "medium".
- HandleRequest.java: **Return response:** The response is standard-compliant but relying on the values of the MIMEConfig and the requested file. Therefore the security threat is considered "very low"
- HandleRequest.java: **Read requested file:** Data is just read and sent to the output but not used any further. This is no direct security threat to the application. While it might be one for the client, that is out of scope.
- HandleRequest.java, other files: **Error output:** This is in the user case valid a HTML error response and otherwise a simple `System.out` statement. This is no security threat.

When we look at this list, we can see all the significant in- and output of the software that interact with the environment. Due to the rather small size of this piece of software, there is not too much I/O; still we could find some problems that might occur. The URI is not validated at all; this could actually be any string without an empty line in it. Thus, this string can contain any coded data and has no length limitation. Since the URI-based file access is done by the operating system, the file access of java will complain in all cases when a file cannot be accessed. Furthermore, due to the use of Unicode, special characters such as quotes etc. should not be a trouble.

Another problem is that the input from the configuration file is not checked. Instead, the code relies on all values being entered into the configuration file. This means that it is possible that attributes are requested though not present. It would be much better if, after the initialization, a check were made that all the required values are initialized. Their validity may be hard to prove, as mentioned earlier. The same problem applies on the MIME types, even though the program here still runs stable, but produces void results if the configuration is void. Since write access to the configuration files gives full control over the server anyway, this access should be reserved for the administrator. The input read from the configuration file could then be regarded as trusted. If, under that circumstance, the server crashes due to bad configuration, it is merely a bug, not a security bug. However, if mistakes in the configuration cause unexpected and unnoticed behaviour, it could be regarded as a security issue (many security breaches are unintentional, and software should be helpful to prevent mistakes). See also the comment on privilege escalation in section 2.4.2.

2.5.3. Evaluation of the Method

This method seems to be good for giving hints for searching for input validation and the associated problems, because it gives specific starting points. This method can also be seen as an add-on to the Threat Scheme, because here, the interaction with the environment is also checked. With the Threat Scheme we focused more on the syntactic validity of the interaction – we cared about what file we could read and if the connection is still alive – now we focus on the semantics and see if the communication follows the standards. So now, we check whether the data in the file is read and interpreted correctly.

As we can see, issues became visible that were still rather hidden in the environment check, for example the fact that in the configuration files properties without values are possible. The drawback of this method is that it also might get rather time intensive if the code has much input-/output-interaction. This is also the reason why it makes sense to put all the input validation in one file/class when developing – the code does not have to be changed in too many places but is instead in one place in the application. The fact that it is a code review should let this part be done quite easily and quickly. This means that less experience is demanded, since the reviewer has to understand what the program does but does not need to know about security issues of specific functions.

2.6. Control Flow Check

2.6.1. The Theory Behind

One issue that is hard to check by any model, checklist or other means is the *control flow* of code. Therefore, we thought it makes sense to make a code review focusing on the control flow. Even if the pure control flow of a piece of code can be modelled nicely with a flow chart, the *processing control* is harder to model. With processing control, we refer to the overall way of how the code is processed. To make this difference more clear we can look at an example in pseudocode:

```
if file X exists
    then delete file X
create file X
```

With the control flow, we mean the `if-then` combination, whereas with processing control we refer to the fact that, in any case, a new file is created and, before that, the old one deleted if it exists. This processing control becomes more important when the application is multithreaded and accesses common sources. Then it may happen that *race conditions* occur. This means that there is a race between two or more different threads, all trying read and write to the same resource, naïvely assuming that the state of the resource is unchanged meanwhile. In reality, unless precautions are taken, there is a small but non-zero chance that the assumption is false. In multithreaded environments, this can occur because threads may be suspended and put to sleep. In our example, these two threads could be the thread running the program and another thread writing to a file, maybe run by the user.

The third part of this check is the error handling and logging. Here it is important that all possible errors are handled and logged, and the program goes back to a stable mode, i.e. it "fails safely".

In this check, we will start out by focusing on race conditions, flow control and other possible timing problems. This should usually be possible to do directly on the code. This means another code review, where this time the main focus is on the question: “Could something bad happen in the time from when I assured myself that it is ok till now?” With some reasoning, we can do this purely on the code; if things get more complicated, we can also use a flow diagram or a sequence diagram. Thereafter we will also have a look at the error handling including the logging of unexpected problems.

2.6.2. Applying the Method

The flow control regarding race conditions is done rather quickly because the biggest part of the program is sequential. There is the point where the threads are forked, but this forking is performed consecutively, so no problems occur there. Moreover, the failure of any action is covered by the code and the system usually fails into a safe mode – even if a certain request then is not processed the server remains online.

One thing that can occur is that the files could be accessed (moved, deleted, etc) between the time they are checked to be directories or to be existent, and the time they are read. In these cases, the reading will then fail and the request will either be answered by an error code or it will just be neglected and some error message be written to `System.out`. This kind of race conditions will always happen, because we always have some time between the check if a file is present and the access on it. Nevertheless, the error handling could have been solved more nicely by a separate *ErrorReportingClass* that does all the logging and reporting of errors. This has some advantages: All error-messages have the same format and are near to each other and thus easy to find. The second point is that if this part has to be changed, then it is really an advantage to have all the output in one place so that everything does not have to be changed.

We also applied the method on the Caudium web server and found problems there. In the standard file system module, filesystem.pike, we found some suspicious lines of code. This code is handling HTTP PUT requests. The following code will remove any existing file with the same name, create a directory if necessary and finally open the file for writing.

```
if(QUERY(keep_old_perms))
  st = file_stat(f);
rm( f );
mkdirhier( dirname(f) );
object to = open(f, "wct");
```

To exploit the race condition, the following must be true: Several users have write access to the same directory; one user tries to upload a file; and another, malicious user creates a file with the same name after `rm()` is called but before the file is opened. The other, malicious user will then gain (or rather, retain) ownership of the file.

2.6.3. Evaluation of the Method

This method has the advantage that it shows the whole construction from another side that was not covered before. Until this method, time was no factor. In fact, some problems only occur in certain situations and constellations. These things have not been regarded before. The problem with these problems is that quite a lot of those just will occur, whether we want it or not. Still, it does not harm to have a look from the more time-dependent point of view.

2.7. Security Principle Checklists

2.7.1. The Theory Behind

According to Viega [3], there are 10 basic guidelines for developing secure software. If these are guidelines for creating secure software and we are searching for criteria to find weaknesses in software, then we think that these rules should also be applicable on security analysing software. These rules are (from Viega [3]):

- Secure the weakest link
- Practice security in depth
- Fail securely
- Follow the principle of the last privilege

- Compartmentalize
- Keep it simple
- Promote privacy
- Remember that hiding secrets is hard
- Be reluctant to trust
- Use your community resources

As each rule is a basic rule for development, these should be regarded in all phases of the development. If this is the case, then the developer probably had security in mind when developing. Moreover, if the developer did so, we should be able to see some of the effort in the work. We try to apply all these rules on our example web servers so that we estimate the overall security.

2.7.2. Applying the Method

When we look at the application we see that it is so small that not all principles may be applied, so we just regard those that can be applied on the software and compare these with the Caudium web server.

Fail securely. This point is quite convenient in Java, because Java forces the programmer to catch exceptions and to fail safely. This is done by resolving the error handling from the program flow, unlike in C/C++ where the errors are indicated with return values. Because of the separation of error handling from regular code, the error handling must not contain program logics, but only cleaning tasks (closing connections, files, etc.). This is done cleanly in the Dahlberg web server. Caudium also fails securely. Scripting errors can for example be printed out, but this traceback is switched off by default for security reasons.

Run with Least Privilege. Because this server should be running on port 80 (according to the configuration), it is difficult to run it without root permissions. The Dahlberg web server stays in the root privilege, whereas Caudium just uses the root privilege to open the listening port, but then runs the scripts and accesses the files as a user with fewer privileges.

Compartmentalize. This was applied in the widest sense because the application was modularized and not written in one big class. Usually this means that the involved systems should be separated from each other and that there should be different levels of authorisation. For this application, it is a generous interpretation of this principle. In comparison, Caudium parses the request in one file, but does all the rest in separate modules which can be loaded and unloaded dynamically (security modules, PHP, user file system, etc.)

Keep it simple. As the Dahlberg web server is a small web server with just few classes and little functionality, it is hard to not keep it simple. The server is implemented like a standard server and the code is easy to read and understand. Caudium is also easy to read from the point of the code, as Pike is really C-like, but to navigate along the method calls and the control structures was harder. This is probably also because of our little experience in programming Pike.

Be reluctant to trust. This guideline was neglected in the request parsing as well as in the configuration part of the Dahlberg web server. It trusts the configuration file as well as the incoming request, which is much worse. Caudium however seemed to be suspicious about the information it got from external entities.

2.7.3. Evaluation of the Method

After checking the code by these more theoretical guidelines, we get a good picture about how much security was on the mind of the people developing. Surely, this is no sure proof that the code they wrote is totally secure, but still it gives us some idea about the level of security. It shows quite easily if these basic principles were applied or not. As this check controls mainly conceptual decisions, it also shows how accurate the review until now was made, in case that new security issues still can be found.

As an easy and quite quick method, this seems to be a good way to give the whole review a final round-up and to provide a good overall picture of the security of the application. As this method is short, if the reviewers are familiar with the reviewed application, we think that this method is still worth the time. However, we also think that the amount of new issues found with this method is rather small.

3. Security Report

3.1. Intention and Structure

At the end of a security review, there should always be a report summing up the result of the review. This report should give the reader specific and descriptive information about problems in the software. This is very important if another application will be based on the evaluated application, because then these security problems have to be kept in mind when using this evaluated part. If this code will be a part of another application, then already in the design of the whole application these overall issues have to be taken into account. By reading the report, it should be possible to get a picture about the software, where its weaknesses could be and where its strengths could be. Still, we can never be totally sure about the security of the tested code.

The structure of the report is a standard report structure with following main points:

1. Title
2. Abstract
3. Introduction
4. Methodology
5. Results
6. Discussion
7. Recommendations
8. Appendices
9. Bibliography

3.2. Example Report

As we are providing the whole example review of a web server written in Java, we also produce the sample report of such an analysis. This will be a sample report for the Dahlberg web server. The layout of this report may differ from any other possible reports, but the result is the real result we got for this web server. As this result should be reproducible, it should ideally also be the same as all other possible evaluation results.

The report can be found in the appendix. (7.3)

4. Summary

As we started to develop this method, we had quite a different view of how these parts have to be linked together and what method should follow which. After a while, we realized the different advantages and disadvantages of each method. One thing that limited our testing experience was the size of our testing target. The Dahlberg web server is rather small and therefore we think that the methods were overkill. On a bigger system, these methods all seem to make more or less sense, because each method shows another point of view for the goal. However, on this small system, they always found the same security issues.

One thing to consider is which methods to use, if there is not so much time for a complete review. If we have to choose which methods we would leave out in a quicker but still thorough review, then we would probably choose a combination of the Design Analysis Check, Attack Tree and the Environment Interaction Check. The reason for this choice is that the Design Analysis really helps to understand the structure of the whole application. Therefore, it does not just only help finding security problems, but this step is also important for preparing for the Attack Tree. Firstly, it generates some useful points to start with; secondly, it is hard to make an Attack Tree for a piece of software, if the software is not known. The Attack Tree is our choice for finding complex security issues – issues, where multiple factors have to play together in the right way for a security hole to open. These leaks are much easier to find through the systematic approach of the Attack Tree much easier to find than by considering what errors could occur at some specific point. The third method of choice was the Environment Interaction Check. The reason for choosing this one is that the care that developers showed in validating input is probably the same care they used for developing the whole package. So if the valid input was not modelled properly, chances are big that errors occur somewhere else as well. The second argument for the input validation is that threats to a piece of software usually come from the outside. Surely, software has bugs, when it is written and error free programming is a main goal, but if a system is attacked, the attack is usually initiated from outside. If this is not the case, the software was not tested sufficiently and probably failed because of errors in the code.

However, we believe that we often do not have a choice when it comes to selecting a method, since it is rather rare that we have everything from documentation to the source code for an application. When we have the code we surely can reverse engineer some models but this also takes quite a long time, and compared to the results from these methods, the effort is probably too big for a code review.

Nevertheless, we think that these methods are helpful, especially to less experienced reviewers, to find criteria for evaluation faster. If a reviewer is experienced, usually she already knows what to search for, but when the reviewer lacks experience, such a review will seem to be a big task, where it is unclear where one should start.

5. Glossary

Analysis tools: A program that goes through the code and tries to find security leaks (unsafe function calls, security estimations for code, etc.).

API: Application Programming Interface; A set of definitions of the ways in which one piece of computer software communicates with another. It is a method of achieving abstraction.

Availability: The property that a service can always be accessed by all authorized users and that it cannot be limited or shutdown by any other instance.

Class loader: An object that is responsible for loading classes. Given the name of a class, it should attempt to locate or generate data that constitutes a definition for the class. A typical strategy is to transform the name into a file name and then read a “class file” of that name from a file system.

Collaboration diagram: A diagram showing the collaboration of multiple entities through their sequential method calls.

Confidentiality: The property that information can only be accessed by authorized users.

Context diagram: A diagram describing the interaction between the environment (stakeholders of a system) and the application.

Control flow: The order how code statements should be executed with given input values, according to the programs algorithm.

Denial-of-service: DoS, an attack on a system forcing it to shut down, to "deny service".

Directory traversal: Getting access on directories that should not be accessible, by tricks with the path, like including double dots ("..") in the path to access higher directories.

Domain model: A diagram describing the domain objects (an entity in a modelled system) that describe business logic and entity relations.

Flow diagram: A diagram describing the control flow of an algorithm.

Integrity: The property that valid information cannot be changed (by accident or maliciously) and also that the origin of the data is the expected one.

Privilege escalation: That a local user or a program is able to gain the privileges of a more privileged user.

Race condition: A race between two different threads. A race condition is present when a value is incorrectly assumed to be constant between two points of execution, when in fact another thread could change the value.

Runtime environment: A software platform (environment) for code to be executed. Usually combined with cross-platform functionality.

Security: Being free of danger that the confidentiality, integrity or availability of a system is decreased.

Security auditing: The process of going through code with an auditing team, trying to find threats and figuring out how secure a piece of code is.

Security testing: Testing code with malicious input to figure out if there are possible security leaks.

Sequence diagram: A diagram showing the method invocations between multiple classes along the time axis.

SQL-Injection: An attack on databases where arbitrary SQL code is executed, that might have been entered through a commonly accessible (web)interface.

UML: Unified Modelling Language, a commonly used, non-proprietary modelling language for analysing and designing software.

6. References

[1] Viega, John and McGraw, Gary. *Building Secure Software: How to Avoid Security Problems the Right Way.* Addison-Wesley Pub. 2001.

[2] Howard, Michael. *Writing Secure Code, 2nd ed.* Microsoft Press 2002.

[3] Viega, John and Messier, Matt. *Input Validation in C and C++.* O'Reilly 2003.

[4] Java security whitepaper, *http://java.sun.com/security/whitepaper.ps.* Sun Microsystems Inc. 1996.

[5] SecurityFocus advisories, *http://www.securityfocus.com.* 2004.

[6] Pike website, *http://pike.ida.liu.se.* 2004

7. Appendix

7.1. Attack Tree

1. Goal: Unauthorized access to information (as the account the server is running as)
    1.1 Simply request any desired file
        1.1.1. Directory traversal attack
            1.1.1.1. Simple directory traversal (/../)
            1.1.1.2. Directory traversal with encoded characters
                1.1.1.2.1. URL encoding (%2E%2E)
                1.1.1.2.2. UTF encodings
                1.1.1.2.3. Other encodings
                1.1.1.2.4. Combinations
        1.1.2. Use a backdoor
            1.1.2.1. – left by developers
            1.1.2.2. – maliciously introduced in a binary distribution
    1.2 Use symbolic links to access files outside the web server tree (local attack)
    1.3 Tamper with the configuration (local attack)
        1.3.1. Trick the server into loading the wrong configuration file
            1.3.1.1. Trick the server into loading the wrong class file (java specific)
                1.3.1.1.1. Exploit bad checking of environment variables
            1.3.1.2. Do tricks with symlinks
            1.3.1.3. Exploit weak file permissions
        1.3.2. Overwrite the configuration file
            1.3.2.1. Exploit weak file permissions
                1.3.2.1.1. Use bad server privileges
                1.3.2.1.2. Use leaks in the operating system
        1.3.3. Change the configuration during runtime
    1.4 Find a valid password
        1.4.1. Guess a password
        1.4.2. Intercept a password sent over the wire (see below)
        1.4.3. Look for passwords in script code
            1.4.3.1. – included in runtime error messages
            1.4.3.2. – accidentally sent instead of executed
2. Goal: Execute code as the web server account
    2.1 Find a buffer overflow
    2.2 Request executable file directly
        2.2.1. Use a backdoor
        2.2.2. Exploit directory traversal to execute any file (see above)
        2.2.3. Exploit bad configuration to execute untrusted code (local attack)
    2.3 Tamper with the configuration (see above)
3. Goal: Access and/or modify other users' information handled by the server
    3.1 Obtain user passwords by intercepting traffic (see above)
    3.2 Guess passwords
    3.3 Tempfile attack (e.g. put symlink with predictable temporary file name in /tmp, pointing to file)
4. Goal: Intercept other users' communication with the server
    4.1 Intercept unencrypted traffic (Outside the scope of this analysis, since HTTP is by design an insecure protocol)
    4.2 Intercept encrypted traffic (SSL/TLS)
        4.2.1. Obtain or replace the server private key
            4.2.1.1. Exploit weak file permissions
        4.2.2. Break the encryption
5. Goal: Cause denial of service (attack on availability)

5.1 Flood the server with requests
5.2 Crash the server
    5.2.1. Request big file
    5.2.2. Send malformed request strings (exploit bad input validation)
        5.2.2.1. Long request strings (buffer overflows)
        5.2.2.2. Un-handled control characters
        5.2.2.3. Zero-length requests
        5.2.2.4. Other types of unchecked input
    5.2.3. Delete executable files
        5.2.3.1. Exploit weak file permissions
    5.2.4. Find a memory leak
5.3 Hang the server
    5.3.1. Request big file
    5.3.2. Find a bug that causes an infinite loop

7.2. Tables

Table 1. Issues found through the Threat Scheme

Problem	Description	Possible Check
Request		
Malformed request	The request could contain invalid characters, random data, it can be malicious or just be almost valid.	Code review (input handling) Security testing (pitfalls, random data)
Big requests	Incoming requests might be longer than standard requests. This might block the application whilst parsing.	Code review (input handling)
Request answered by the wrong process	If incoming requests are distributed either by the OS according to the socket or by the program according to the request string, a wrong decision can be taken.	Code review (input handling, control forwarding)
Response		
Responding to the wrong address	The response is sent to another than the requesting address.	Code review (tampering with answer address)
Too big response	Resources can be blocked if too big resources are requested (e.g. network throughput)	Code review (response length & handling)
Response contains wrong data	The response contains non-requested information like debug info, or other data.	Code review (response handling)
Resource Access		
Changing system configurations/limitations	Limitations by the JVM could be changed, also write access on system files might happen.	Code review (request handling) Attack Tree (access on system files)
Illegal access of resources	Can resources like network, databases or printers accessed or can the access to these be hijacked?	Attack Tree
Executing files	Is it possible to execute files. Are these files in some specific folder that cannot be accessed otherwise. Can files be executed, which should not be executable?	Attack Tree.
Overloading resources	Is the access to resources (CPU, RAM,...) limited or unlimited?	Code review (thread handling)
File system Access		
Illegal file-access (system or configuration files)	Is it possible to access any other files, that are not in the web-root?	Attack Tree.
Illegal file modification	Is it possible to modify somehow files that should not be modifiable?	Attack Tree.
Interference with files and file system	Is it possible to interfere with the file system in terms of deleting, moving, etc files?	Attack Tree.

Table 2. Design Analysis results

Threat	Description	Planned/possible behaviour	Possible check
Wrong configuration	The wrong file could be loaded or the	Server is executed but behav-	Attack Tree

Threat	Description	Planned/possible behaviour	Possible check
file	file is not found.	iour is uncertain	Review code
Wrong configuration values	Values in the configuration file can be wrong.	Server is executed but behaviour is uncertain.	Review code Check the configuration file documentation
Wrong MIME file	The wrong file could be loaded or the file is not found.	Server is executed but behaviour is uncertain, probably void MIME-type in HTTP response.	Review code Check the MIME-file documentation
Wrong MIME values	Values in the configuration file can be wrong.	Server is executed but behaviour is uncertain, probably void MIME-type in HTTP response.	Review code Check the MIME-file documentation
Starting fails	The startup of the server fails or the server-socket cannot be opened.	Sever is not running. Resources might be blocked.	Review code (occupied resources)
Connection dies	The established connection suddenly dies.	No response is sent	Review code (error handling)
Multiple connections in short time	A big amount of connections are opened at the same time, try to connect at once.	Serves one after each other and slows down.	Review code (thread handling) Security testing
Starting the handler fails	The request handler cannot be started.	Request is not served, server runs on	Review code (error handling)
Wrong handler started	Another than the appropriate handler is started.	Server behaviour is unpredictable	Attack Tree Review code (flexible class for name possible in configuration)
Output-stream is not ready	The connection is established, but nothing can be written into the output-stream.	Request not served	Review code (stream handling)
Invalid request	The request is invalid (contains special characters, malformed, etc).	Only valid requests are processed, in worst case all requests are processed	Review code (request parsing)
Parsing has wrong result	The result of parsing the request is not the result that should be created.	Wrong result is processed	Review code (request parsing)
Request too big	The request-string is too long and blocks the request-parser.	Request is tried to be handled, in worst case the application runs out of memory	Review code (request limitations)
File not available	The requested file is not available	The request is not served	Security testing (request non-available file)

Threat	Description	Planned/possible behaviour	Possible check
Illegal file requested	A file is requested, that should not be accessible.	File should be checked, request is served in worst case	Review code Security testing (directory traversal)
Index not available	For directories is no index files available,	Directory listing is generated	Review code
Requested file is not ready	The file that should be served is still locked and not ready for being read.	Program waits until the file is readable, response may be delayed.	Review code Security testing
Requested file is illegal file	The file requested is not valid (contains invalid characters or checksums, etc)	Request not served	Review code Security testing
Requested file not readable	The requested file cannot be read because of encryption, encoding, etc.	In the worst case the encrypted text is sent back.	Security testing
Requested file too big	The requested file is too big for sending easily.	The server sends the file so network resources may be blocked, in worst case the server runs out of memory.	Security testing

Table 3. Environment Interaction Check results

File	Line(s)	Action	Check	Severity
MIMETypes.java	27 - 51	Read configuration	Input cannot be validated since almost anything can be valid input. Void input does not harm the application but sets some wrong parameters in the response. Properties without Value are possible.	Very Low
ConfFile.java	10 - 50	Read configuration	Input cannot be validated because almost anything can be valid input. Void input can harm the application in making it crash or not find the needed data. Properties without Value are not possible. No check for a valid configuration.	Low
HandleRequest.java	64 - 78	Parse request	Basic parsing because just partly used. The read URI is not validated.	Medium
HandleRequest.java	126 - 172	Return response	The response is standard-compliant but relying on the values of the MIMEConfig and the requested file.	Very Low
HandleRequest.java	144 - 149	Read requested file	Data is just read and sent to the output but not used further.	None
HandleRequest.java	183 - 194	Error response	Returns standard compliant data.	None
HandleRequest.java	198 - 213	Error response	Returns standard compliant data.	None

Various files	N/A	Error-output	Simple System.out statements that do not have to have any specific format.	None

7.3. Example Report

1. **Title**

Security Review of the Dahlberg Web server, version 2 - May 10, 2004

2. **Abstract**

This report gives a security overview about the Dahlberg web-server written in Java. It shortly describes the methods that were used to find any security problems in the web server. It also describes the problems found and recommends possible actions to solve these problems, if the web server is used in a bigger context. The major problems that were found were directory traversal, local file access issues, input check validation problems and denial-of-service vulnerabilities.

3. **Introduction**

The Dahlberg web-server from *http://www.dahlberg.se/java/httpd/* is a basic, standard-compliant web-server completely written in Java. Even though its functionality is quite limited – it just serves the HTTP GET request – it is still a possibility for a small and basic web-server running on the local host. It supports GET requests, directory listings, index files and basic error responses. It is configurable and can be run by any user on a higher port-number (> 1024) or as root on any possible port. The web server is written in four classes and uses two configuration files to keep some parameters flexible. One of the configuration files is for the general server configuration; the other one is for configuring the MIME type in the HTTP-responses.

4. **Methodology**

The applied methodology follows the scheme described in the review paper of Holmgren and Mähr [a]. This approach is a top-down approach on existing software with a security review of the code without taking influence on it. We started with a Threat Scheme on this application to model the software in its environment. Then we did a Design Analysis to have a closer look on the design of the software and possible problems in there. Thereafter we made an Attack Tree analysis on the web-server for finding more complex security issues. These checks were then followed by two code reviews with focus on different aspects. At first we focused on the direct environment interaction, and after that we continued by going through the program analysing the control flow and finally we made an overall estimation of the general impression of the code security.

5. Results

When we were applying these methods, we found following security issues:

- Directory traversal is possible.
- With local file write access the server can be forced to execute random code or just crashed.
- Possibility to overload the server with requests.
- Files of any size are served.
- Request are also processed if the are malformed and not standard-compliant.
- Input validation of configuration files is sometimes weak.

6. **Discussion**

These points are all security problems we encountered in the web-server. From these, the biggest problem is posed by the directory traversal because through this leak anybody can easily request any local file. As the Dahlberg web server is run on port 80 by default, and root privileges are needed for opening this port and these privileges are never given back, the software runs as root. This alone is bad, because in case of failure or in case of a successful attack on the server by a malicious user, the user could use this as privilege escalation. In combination with the possibility of a directory traversal (the document path including the two dots is gladly used), it is possible to read every file on the whole system. This includes all the root-only readable files that store passwords, but it also includes files from other users, which could contain sensitive data. Through this leak, it should not be too hard to read user information like username and passwords (or their hashes) for gaining shell access to the server. This is a real and big threat, opening a big security hole in any secure system.

A much smaller problem is the local access. If someone manages to access the program files and manages to replace the `HandleRequest.java` file with his own version, then this one also can execute arbitrary code with the same rights as the rest of the application. Another possibility is to tamper with the configuration files, which then at least makes the server unavailable or makes it generate useless output. Fortunately, the configuration files are not too powerful, they do not define, which classes are called are loaded, or the paths to used classes, but they still define the document root. In this case (through the problem of directory traversal - see above), this does not make too much difference anymore.

The fact that the server handles every request, independent on the size of the requested file, makes it vulnerable against denial-of-service attacks. If the document root contains bigger files, then they are served. Furthermore, the server handles each request. Although the length of the request string is not threatening to – too long requests do not block the server – the server can be slowed down significantly if a big amount of requests is sent to it at the same time. Ten threads are by default waiting for requests, but every request more than ten has to wait until a request handler has started up, which may take some time.

The fact that the configuration is not validated while being loaded conceals some possible problems. If something goes wrong while loading these values, the main program may not realize it and may run with just the half of the values set properly, as it is also possible to just have parts in the configuration without any value, with just a key. At least the availability of all keys should be checked before they are used.

The processing of non standard-compliant requests is actually no real security threat, in this time it is even an advantage because the server is not vulnerable for any attack witch long requests.

7. **Recommendations**

Considering the directory traversal problem we recommend strongly to change the code (as far as it is legally possible), this check is just a minor change in the code and does not take too much time. Especially in combination with the root rights, this leak is too big and too easy to use. The second thing that should be taken care of, when using the Dahlberg web server, is to ensure that no non-privileged user has any access to the program files of the web server, not access to the configuration files. To deal with the possible threat of a denial-of-service attack, it would help to at least limit the size of outgoing files. This means that before serving the files, the server checks their size and just delivers them, if they are small enough or if there is not too much other traffic. Against the fact that there could be a big amount of clients connection at the same time cannot be done too much, the ten request handlers that are waiting should be enough and for normal use it does not make sense to have more of them waiting. For the issue with the configuration files we also propose to change the code so far, that after reading the files, the program validates, if there are valid values in the configuration and otherwise refuse to start. This requires some modelling of possible good configuration data. If these issues are covered, the server can safely be used for small web serving. There still might be security issues hidden in the application, but compared to the size of the application and the frequency of use of this web server it seems to us that the server has been audited satisfyingly.

8. **Appendices**

None. *Here would be the copies of the results of each method (i.e. table.1) but for size reasons we leave them out here*

9. **Bibliography**

[a] Holmgren, Magnus and Mähr, Wolfgang. *Applied Security Auditing.* Linköping 2004.