DEVELOPING SECURE SOFTWARE

A survey and classification of common software vulnerabilities

Frank Piessens
Dept. of Computer Science
Katholieke Universiteit Leuven, Belgium
Frank.Piessens@cs.kuleuven.ac.be

Bart De Decker
Dept. of Computer Science
Katholieke Universiteit Leuven, Belgium
Bart.DeDecker@cs.kuleuven.ac.be

Bart De Win
Dept. of Computer Science
Katholieke Universiteit Leuven, Belgium
Bart.DeWin@cs.kuleuven.ac.be

Abstract More and more software is deployed in an environment with wide area network connectivity, in particular with connectivity to the Internet. Software developers are not always aware of the security implications of this connectivity, and hence the software they produce contains a large number of vulnerabilities exploitable by attackers.

Statistics show that a limited number of types of vulnerabilities account for the majority of successful attacks on the Internet. Hence, we believe that it is very useful for a software developer to have a deep understanding of these kinds of vulnerabilities, in order to avoid them in new software. In this paper, we present a survey and classification of the most commonly exploited software vulnerabilities.

Keywords: security, software vulnerabilities, software engineering

M. Gertz et al. (eds.), *Integrity, Internal Control and Security in Information Systems*

1. Introduction

At the root of almost every security incident on the Internet are one or more software vulnerabilities, i.e. security-related bugs in the software that can be exploited by an attacker to perform actions he should not be able to perform.

Experience shows that a majority of these software vulnerabilities can be traced back to a relatively small number of causes: software developers are making the same mistakes over and over again. Looking for instance at the list of the ten most exploited software vulnerabilities (see: [11]), one can see that many of these vulnerabilities are actually buffer overflow problems.

Hence, we believe that it is useful to try to survey and classify the most frequently occuring types of vulnerabilities. This paper identifies a number of categories of software vulnerabilities, and gives extensive examples of each of these categories. A software engineer familiar with these categories of problems is less likely to fall prey to these same problems again in his own software.

The structure of this paper is as follows: in the next section, we present a structured classification of software vulnerabilities. In section 3, we present a number of easily remembered guidelines a software developer can keep in mind to steer clear of most of the identified categories of problems. We conclude by discussing related work and summarizing our results.

2. An illustrated survey of software vulnerabilities

2.1 Insufficiently defensive input checking

A developer regularly makes (often implicit, and at first sight very reasonable) assumptions about the input to his programs. An attacker can invalidate these assumptions to his gain. It is important to realize that *input* should be interpreted in a broad way: input could be given to a program through files, network connections, environment variables, interaction with a user etc... If method calls in a program can cross protection domains (as is the case for instance in Java), even method parameters should be considered as non-trustworthy input that needs careful checking.

Examples of this category include: buffer overflows and weak CGI scripts.

2.1.1 Buffer overflows. One of the most successful attacks is certainly a buffer overflow in a server process. What happens is illustrated in figure 1. For every function call, the parameters, the return address[1], and the local variables of the function are put on top of the stack, the so-called current stack frame. In figure 1.a, the normal situation is sketched. The return address points to the correct instruction. If the function is not carefully programmed and does not take into account the actual sizes of the variables, then a buffer overflow might happen. For instance, many servers expect input from client processes, such as a name, a path, an email-address, etc. Often, the server has allocated a buffer for this input in the current stack frame. The buffer is usually oversized and is certainly large enough to accommodate all 'reasonable' input. However, if an attacker sends input that is much larger than expected, and if the server does not take appropriate precautions (e.g. it copies the input until a zero-byte is found), then part of the stack frame is overwritten (see also figure 1.b): the return address is modified and points to code that has been sent as part of the input! Hence, the attacker can make the server execute whatever code he wants to. Usually, the code that is sent as input will make the server spawn a new process that runs the command interpreter (the shell). That way, the attacker gets inside the system without a login procedure. Often, he has superuser privileges, since the command interpreter inherits the privileges of the attacked server.

The last decade, many server programs have been found to be vulnerable to this kind of attack (the finger daemon, bind daemon, ...). Often, these servers used a getstring function that did not limit the length of the string to be read. However, newer attacks do not use 'oversized' input, but cause buffer overflow by other means. For instance, inputs with special characters (wildcards, ...) are sometimes expanded (globalized) by the server, leading to buffer overflow. Also, incomplete inputs may divert (temporarily) the flow of control inside the server; after this diversion, assumptions about the sizes of the variables may no longer be true.

2.1.2 Weak CGI scripts. CGI, an acronym for Common Gateway Interface, is a mechanism for extending the webserver. Instead of sending a webpage in response to a client request, the server starts a new process which will handle the request. Typically the subprocess runs a script (e.g. Perl, Tcl, ...). These languages offer pattern-matching

[1]The return address is the address of the instruction that must be executed after the function call.

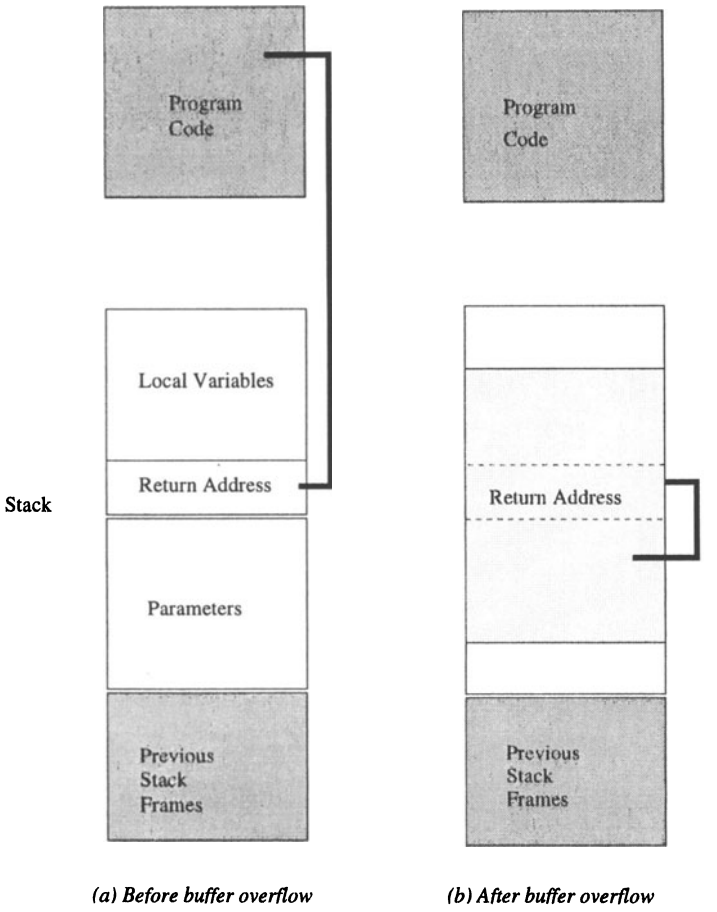

Figure 1. Buffer overflow attack

and many other features, that are useful in this context. However, the expressions are even more powerful than those supported by command interpreters (shells). Since the user inputs are passed to the script, insufficient checking of these inputs may lead to disaster. Figure 2 illustrates the CGI-mechanism. Often, CGI scripts are used to process 'forms' included in certain web pages. The users completes the input fields and submits the form to the server. The server spawns a new process that runs the script, and passes the user inputs either through environment variables, or via standard input. Assume that one of the inputs is an

email-address, that will be used to send a confirmation message to the user, and that the Perl script contains the following lines of code:

```
⋮
$emailaddress = ...;  # fetch the email address
⋮
system("echo \"Your form has been processed\" | mail $emailaddress");
⋮
```

If the email address is not checked by the script, then a user has the ability to have the script execute whatever command the user wants. In this case, if the user gave as email address: `user@dot.com; rm -rf /` then, eventually the following commands will be executed:

```
echo "Your form has been processed" | mail user@dot.com; rm -rf /
```

that is, the acknowledgement is sent to the user, and the file system may be wiped out if the web server is running with superuser privileges! Instead of this denial of service attack, the malicious user could also try to add an extra line to the password file and thus create for him an entrance gate to the system.

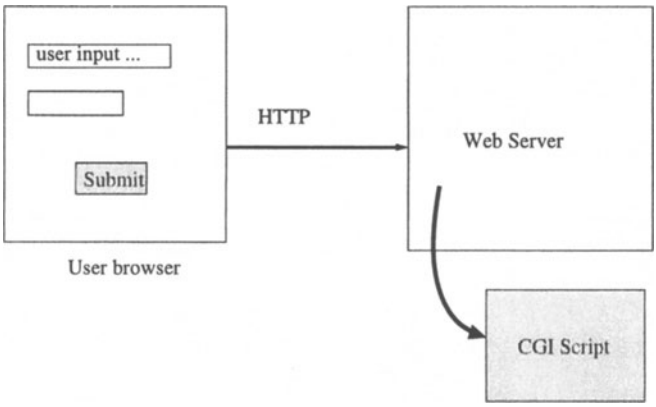

Figure 2. CGI Scripts

2.2 Reuse of software in more hostile environments

Software written for use in a relatively friendly environment (like a mainframe or an intranet) is often reused later in a more hostile en-

vironment (like the Internet). Because the software developer made certain assumptions about the environment in which his program would be running, this change in environment can lead to major security holes. Typical examples include programs using password authentication, and document processing software reused as content viewers on the Internet.

2.2.1 Password authentication. Under the assumption that passwords are well chosen, and well guarded by their owners, password authentication is relatively secure in an environment where the communication between the user typing in the password and the computer verifying the password can not be eavesdropped on by attackers. Typically, for a terminal connected to a mainframe by a dedicated line, a password mechanism is sufficiently secure.

However, in a context where the password is communicated over the Internet, password authentication is extremely weak: eavesdropping on connections is commonplace on the Internet, and once a password has been seen by somebody else, the security of the mechanism is completely broken. Still, many popular programs like telnet and ftp rely on this mechanism to authenticate connections over the Internet.

2.2.2 Document processing software reused as Internet content viewer. If word processing software is used to create, edit and view documents authored by the owner of the software, or a small set of trusted colleagues, then the security requirements of this software are relatively low. If the software contains buffer overflow problems, it might crash occasionally, but it does not represent a major security problem.

This situation changes completely if the same word processing software is reused as a viewer for Internet content. The same buffer overflow problem can now be maliciously exploited: an attacker places a carefully constructed document on the Web, and tries to lure victims into viewing this document. By exploiting the buffer overflow problem, the attacker can do anything he wants on the victims computer.

Since it is difficult to predict in advance in which contexts your software will be used, it is good practice to strive for secure software development even for software that will initially only be used in a friendly environment.

2.3 Trading off security for convenience or functionality

It is well-known that there is a trade-off between security and convenience (i.e. functionality or user-friendliness of the software). Most

security measures tend to add some user-annoyance, and often very powerful and convenient features are easy to abuse. Software developers, tending to think of functionality in the first place, usually emphasize convenience over security. This problem is often an attitude problem: software developers tend to spend a lot of time thinking about how to make things possible. From a security point of view it is as important to spend time thinking about how to make certain things *impossible*.

Examples of vulnerabilities in this category include: executable attachments and powerful scripting languages for applications.

2.3.1 Executable attachments. Many browsers maintain a table which is used to determine how the browser should handle MIME types when it encounters MIME parts in a HTML document, be it an email message, a newsgroup posting, a web page, or a local file. Some of these entries may cause the browser to open the MIME part without giving the end user the opportunity to decide whether the MIME part should be opened. Hence, an intruder may construct malicious content that, when viewed in the browser (or any program that uses the browser's HTML rendering engine), can execute arbitrary code. It is not necessary to run an attachment; simply viewing the document in a vulnerable program is sufficient to execute arbitrary code.

2.3.2 Powerful scripting in applications. More and more applications include an interpreter for a scripting language, which can be used to support 'dynamic' content. Examples are word processors, spreadsheets, web browsers, etc. The problem with these scripting languages is that they are very powerful, and often allow access to local system resources, such as the file system. Although a technique, called sandboxing, can shield off the local system, dynamic content can still mislead the user, and possibly capture confidential information, such as credit card numbers, passwords, etc.

The following attack against web browsers that support JavaScript has been described by Felten, Balfanz, Dean and Wallach ([5]). See also figure 3. An unsuspecting user is lured to the attacker's website[2]. The web document shown to the user is 'booby-trapped': it contains a JavaScript program, which disables the normal functioning of the browser's buttons (by covering the browser with an invisible window), and all URLs are rewritten in order to direct requests to the attacker's site. From now

[2]This is probably the easiest part of the attack. It suffices to offer something for free, to attract many possible victims.

on, the browser is actually captured by the attacker. There is no way to escape. Every URL in the document is of the form:

`http://www.attacker.com/http://www.real.com/page.html`

Hence, the request is sent to `www.attacker.com`, which will forward to request to `www.real.com`. The web document that is returned by that server is then rewritten by the attacker's website, i.e. all URLs are rewritten and a JavaScript program is added to the document. The modified document is finally sent to browser. Note that the JavaScript program can hide these modifications to the user: if the browser is asked to show the 'HTML source' of the document, the script will remove the malicious code and show the original URLs. Since all requests are sent to the attacker's website, including input fields of forms, the site may acquire and abuse confidential information.

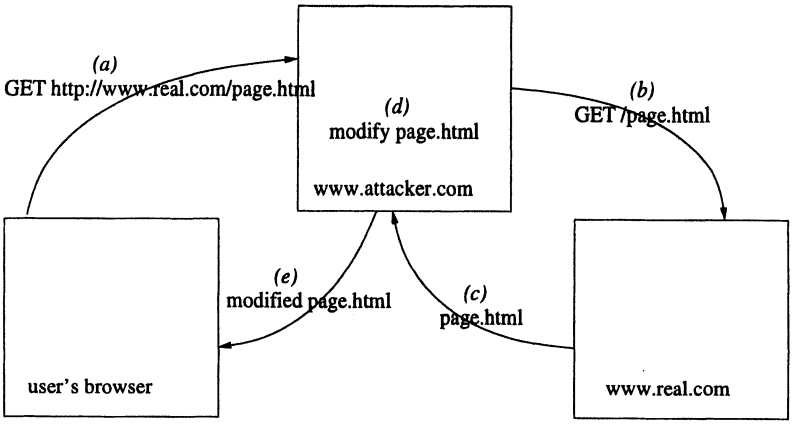

Figure 3. A webspoofing attack

2.4 Relying on non-secure abstractions

Many abstractions offered by a programming language or by an operating system are "complexity-hiding" abstractions more than "tamper-proof" abstractions. Software developers often (implicitly or explicitly) assume that these abstractions are tamper-proof anyway, leading to security breaches. Examples include: buffer overflows (see section 2.1.1), type confusion problems in Java, attacks against smartcards, considering TCP/IP connections as reliable communication channels, unanticipated object reuse,etc... We discuss two examples in more detail.

2.4.1 Type confusion in Java. To allow untrusted code limited access to objects, it seems reasonable to use object oriented access specifiers (like private or protected) on methods or fields that should not be accessible to the untrusted code. The programmer relies on the information hiding aspects of the object oriented language he is working in to achieve a security related goal.

However, it is important to realize that access specifiers are by no means "tamper-proof" in most object oriented languages. For example, in C++, untrusted code can scan the entire memory range in use by a process by casting integers to pointers, and hence untrusted code can also access the private fields of any object. In other words, the OO abstractions offered by C++ are not secure, or C++ is not memory safe or type safe.

The designers of Java tried to make the Java Virtual Machine memory safe and type safe by disallowing pointers, and by checking casts even at runtime. But for many versions of the JVM, bugs in the implementation of the VM have led to breaches in memory and type safety. For example, the well-known classloader-attack (see: [8]), breaks the type safety of all JVM's upto version 1.1.8.

Even in the absence of type safety problems, untrusted code may try to access private fields of an object by serializing the object, and reading the resulting file as a byte array.

2.4.2 Unanticipated resource reuse. The problem here is that the software developer disposes of some object or resource (e.g. deletes a file), and assumes that by disposing of the object, its information content becomes inaccessible. In many cases the object will only be "logically" deleted, and the actual content is still retrievable by an attacker.

2.5 Insecure defaults and difficult configuration

The default configuration of general purpose software is often not secure to guarantee that the majority of customers is able to use it without experiencing too many restrictions. Especially for operating systems, this is common practice. Clearly, the users impression about the system is important and security restrictions might annoy him. However, this should not be a reason to lower the level of security or it should be explicitly and very well documented. And even in this case, system administrators typically do not take the time to read this documentation. They tend to make a default install, and if that works leave it at that. Hence, if the default configuration is an insecure one, many installations will be in an insecure state.

As an example, Microsoft Windows NT 4.0 is a reasonably secure operating system as proven by the ITSEC E3/F-C2 label it received after independent evaluation. However, a default install of the system disables many of the security features. Several documents (see: [4, 2]) provide checklists of tasks an administrator should perform to enable important security features and as such augment the overall security level of the system. However, it is highly questionable how many users will follow all the guidelines described in these checklist documents.

As another point of attention, configuration procedures are sometimes complex and error-prone. For example, securing Windows NT requires changing certain keys in the registry by editing them by hand. Complex configuration procedures must be avoided, since they lead to configuration errors, and a configuration error often introduces a security problem.

2.6 Unanticipated (ab-)use of services and feature interaction

Highly successful services are often used (and abused) in ways never imagined by the designers of the service. Hence, the designers failed to provide safeguards for these abuses.

A typical example is e-mail. The Internet e-mail system, based on SMTP, was designed to provide a simple electronic messaging service for a relatively limited group of people. The unforeseen success of TCP/IP and the Internet has made SMTP a standard for a worldwide electronic mail system. Since sending e-mail is typically much cheaper than sending paper mail, advertisers have been abusing the e-mail system since many years, sending out advertisements to millions of addressees at once. Because the designers did not anticipate the enormous success of their protocol, they did not think of safeguards for protecting against such spam e-mail.

A special case of unanticipated abuse is *feature interaction.* As more and more features are added to a software product, they start interacting in unforeseen and insecure ways. An example is the telephone network, where the introduction of new services, like call-forwarding, conference calls and ringback have led to numerous security breaches ([1]).

2.7 Non-atomic check and use

A typical scenario in a security relevant part of a program is: check if some condition is ok, and if it is, perform some action. Often attacks are based on invalidating the condition between the check and the action.

A typical example is a so-called *race condition.* For example, a program checks to see if a certain filename in the temporary directory is

available (i.e. no file with that name exists already), and if it does not exist, it opens a file with that name and starts writing information to it. An attacker can try to create a link with that specific name to a file he wants to alter between the check of existence and the actual opening of the file. As a consequence, the attacker causes the program to inadvertently add information to an existing file, where the program tried to enforce that it was really opening a new file.

A second, very simple example is simply typing commands at an unattended terminal: the operating system only checks the identity of the user at login-time, and from that moment on assumes that all commands from that terminal come from the authenticated user. A similar problem occurs with session hijacking of telnet sessions over the Internet.

2.8 Programming bugs

Finally, ordinary programming bugs, i.e. flawed algorithmic logic in security sensitive software, are much harder to detect during testing than bugs in the functionality of the software. Security related bugs only show up in the presence of malicious adversaries and hence can not be detected using automatic testing procedures. Moreover, the inherent complexity of cryptographic algorithms and other security related code makes it very hard to understand all relevant details and unfortunately wrong assumptions or small programming errors often introduce big security holes. Several famous examples of this problem exist. First, a weakness in the random generator of Netscape 1.1 where random numbers where based on the current time (which is not random at all !), made it possible to break the keys used in secure connections within seconds (see: [9]). Another example is known as the Java DNS bug. Here (see: [3, 6]), an error in the algorithm used to check whether two hosts are equal provided applets with the opportunity to connect to every computer on the Internet, which was not conforming to the rules of the restricted applet execution environment.

3. Security guidelines for developers

Many of the example software security weaknesses discussed in the previous section could have been avoided if the designers and implementors of the software had been more security-conscious during their design and programming. We feel it is very important for a software engineer to keep a number of security-related design guidelines in the back of his head at all times during the development of a software system. Every design or implementation decision should be verified against these "security rules of thumb". A good set of such guidelines is given below.

It is interesting to note that these guidelines still overlap significantly with the guidelines given in the 25 year old classic paper by Saltzer and Schroeder ([10]).

1 *Defensive programming.* Treat any input your software gets from outside as potentially hostile.

2 *Secure defaults.* While it may be a good idea to make security-related parts of your program configurable, you should realize that many users will use the default configuration without thinking too much about it. Hence, the default configuration should be secure.

   Also, many security checks can be implemented in two ways: deny by default and allow access in selected cases, or allow by default and deny access in selected cases. It should be clear that the first approach is preferable.

3 *Use secure languages where possible.* From a security point of view, a garbage-collected language (like Java) is to be preferred over a language relying on manual memory management (like C or C++). In particular, a type safe language significantly reduces the number of potential security weaknesses in software.

4 *Security-oriented testing.* Software engineers should realize that testing for security is fundamentally different from testing functionality. Testing for security is a creative form of testing, where the testers have to come up with possible attack scenarios.

5 *Economy of mechanism.* Security mechanisms should be as simple as possible (but not any simpler than that). A simple mechanism is easy to understand, easy to verify, and easy to apply.

6 *Need to know principle.* If it is possible to give different parts of your software different privileges (as is possible in Java for example), make sure that you give each part the minimal amount of privileges necessary. This leads to better containment of security breaches.

   As another instance of this rule: make sure your software can run with the minimal amount of privileges from the OS it is running on. Software written for Windows 9X for example, typically assumes having full access to the entire file system, making it difficult to port this software to the more secure NT family of operating systems.

7 *No security decisions by end users.* End users typically have little or no expertise in security, and asking them to do security relevant

configuration easily leads to configuration errors. Also, attackers might try to convince end users to change their configuration to a nonsecure state through social engineering techniques.

4. Related Work

An influential paper surveying and categorizing software vulnerabilities is the paper by Landwehr et al. ([7]). However, this paper is largely focused on system software vulnerabilities, whereas our paper mainly targets application software. A number of books ([1, 6]) give many examples of vulnerabilities, but without an attempt at classification. Also many websites publish lists of software vulnerabilities of varying quality. A column by McGraw and Viega on the IBM DeveloperWorks website is of very high quality ([12]). Finally, our security guidelines were heavily influenced by the seminal paper by Saltzer and Schroeder ([10]).

5. Conclusion

A classification of the most common software vulnerabilities (with many examples) was presented. This classification shows that many software vulnerabilities can be avoided by keeping in mind a number of simple security-related guidelines during design and development of software.

References

[1] Anderson, Ross (2001) *Security Engineering. A Guide to Building Dependable Distributed Systems.* Wiley and Sons publishers.

[2] Paul F. Bartock et al., Guide to Securing Microsoft Windows NT Networks, *National Security Agency*

[3] DNS based attack on Java, *http://www.cs.princeton.edu/sip/news/dns-spoof.html*

[4] Micheal Espinola Jr (Santeria Systems), The Hardening of Microsoft Windows NT, `http://www.networkcommand.com/docs/HardNT40rel1.pdf`

[5] Edward W. Felten, Dirk Balfanz, Drew Dean, and Dan S. Wallach, "Web Spoofing: An Internet Con Game", 20th National Information Systems Security Conference (Baltimore, Maryland), October, 1997.

[6] Gollmann, Dieter (2000) *Computer Security.* Wiley and Sons publishers.

[7] CE Landwehr, AR Bull, JP McDermott, WS Choi, "A Taxonomy of Computer Program Security Flaws, with Examples", ACM Computing Surveys 26, no. 3 (Sep 1994).

[8] Sheng Liang, Gilad Bracha, "Dynamic Class Loading in the Java Virtual Machine", Proceedings of the Conference on Object-oriented programming, systems, languages, and applications (OOPSLA'98), pp. 36 - 44.

[9] Netscape (In)Security Problems, *http://www.demailly.com/ dl/netscapesec/*

[10] Jerome H. Saltzer and Michael D. Schroeder. "The protection of Information in Computer Systems", in *Proceedings of the IEEE*, vol. 63 no. 9 (Mar 1975), pp. 1287–1308.

[11] SANS Institute, The ten most critical security threats, `http://www.sans.org/topten.htm`

[12] http://www.ibm.com/developerworks/security/