

# Software Bugs: Distribution, Exploitation and the Future

Felix 'FX' Lindner, Recurity Labs GmbH, fx@recurity-labs.com

## Introduction

Significant research efforts were undertaken in the last years to detect and prevent software vulnerabilities from being exploited to gain unauthorized access to computer systems. But actually very little is understood of the nature and classification of those software faults that are actually used to break into systems. This leads to valuable resources being spent on projects with inadequate goals to downright wrong ideas [1].

Before researching prevention and countermeasure techniques, it is necessary to obtain and interpret solid data on how most computer systems are compromised and what types of faults were used on the way. They are the ones the research must concentrate on and they are not necessarily the ones most often talked about. Also, it is very important to understand the difference between a bug, a vulnerability and an actual exploit. Attempts to prevent the later will often be easily circumvented because the prevention technique did not take the differences into account.

In this article, the term Hacker is often used to refer to a non-academic security enthusiast, who might or might not work professionally in the computer security industry. In this context, I refer to the limited subset of the security community who try to understand the broader meaning of a vulnerability, develop new mechanisms to exploit them and generally perform what the community calls "research", but which has more to do with experimenting and creative application.

The software testing community and with it the academic groups working on reliable and dependable software are in general concerned with the same issues in software that hackers and security people have an interest in. If the software would work exactly as specified or intended by the maker, only a small subset would be still attackable and defences would be much easier to implement. But as of today, almost all software shows failures, the well-known software bugs.

The hacker and security community has developed it's own terminology for phenomena that are named differently in software testing, since software testing names issues by their root cause, while hackers name issues by bug classes. A bug class is a type of software defect of which the community knows how to exploit it. All other software faults are simply classified as functional or not security relevant bugs. When a specific type of coding fault is first exploited on a system, it becomes a bug class and instances of the same type of issue are searched for in all other software. The software fault used in an exploit does therefore always belong to a bug class, while most software faults don't, since it's not yet known how to exploit them.

When looking at the academic naming of software defects, many of the obscure names from the hacker community become clear: Buffer Overflows and Integer Overflows are named Data Reference Failures in software testing. Also, most so-called Denial-of-Service problems with software are actually Data Reference Failures, some of them being falsely classified as "just DoS" and not correctly named non-exploitable or not exploited Buffer Overflows. This is another indicator that the naming is actually based on exploitability rather than cause. So-called Format String bugs and several types of Race Conditions belong to the class of Interface Failures. Directory Traversal bugs, Illegal Directory or File Access as well as Remote

Command Execution are also in their waste majority Interface Failures. The difference here is, that the interface failing is so important that the author thinks they deserve their own name: Operating System Interface Failures. Two bug classes that emerged later than the others are SQL Injections and Cross Site Scripting vulnerabilities, which both can be classified as Input/Output errors. Academic reliable software and software testing research knows several more defect types that the hacker community is not (yet) exploiting and hence hasn't named them.

The different naming conventions are often a source for massive misunderstanding between the academic and the hacker world and have probably led to a several misinterpretations of actual data. Apart from software failing to implement authentication and authorization correctly, which is referred to as Logic Flaws in the community, every security hole is a software fault in the scientific sense.

## Turning a malfunction into an exploit

How do hackers distinguish between a software bug and a vulnerability? The transition from a bug to a vulnerability is a core concept everyone working on computer security must understand to create appropriate defences. As an example, finding all potential buffer overflows in a given piece of software might yield a number of issues, of which none, some or all could be classified as vulnerability. The classification depends on *who* can cause the condition, *what* is the effect of the condition occurring and *how* can it be used to perform an action that would normally be restricted. None of these questions are easily answered in today's heterogeneous and complex environments.

The "who" question is the most obvious of the three. If the software in question runs on a networked computer and an anonymous user on the same network can cause the error condition, it qualifies automatically as vulnerability, since the anonymous user has by the definition of authentication and authorization a lower privilege than the context the software runs in. The same holds true if the software runs with a different privilege context on the same machine as the attacker has access to, since the subject (attacker) should not be able to influence availability, confidentiality or integrity of another privilege context.

When there is no obvious way for an attacker in a different privilege context to trigger the error condition, it often leads to the assumption that the fault is not a vulnerability. Typical examples are faults in command line argument handling of UNIX command line tools. While an attacker in a shell environment can still causing the condition, the faulty program would run in his privilege context and therefore a successful attack would only yield the privileges the attacker is already entitled with. But when a web based program runs the same faulty software, the attacker has a much lower privilege, because he should not be allowed to run arbitrary code on the web server.

The "what" question and the "how" question are somewhat related while not completely the same. When a software fault is discovered and can be triggered from a lower privileged context, the hacker will observe closely what happens in case of the error condition and especially how much of the unexpected reaction of the software can be deterministically used. One could call this part of the process the creative creation of functionality from where was none before. The general method on how to exploit a faulty behaviour is also the point where the software bug will be classified into one of the bug classes. Examples on how this is done will be shown later in the article.

Finally, the “how” question decides on the actual implementation of the attack. Here, the hacker almost follows a development procedure, since the environmental parameters as well as the available tools are known at this point in time. Given these, the hacker will develop an actual attack, often by writing a program himself that carries out the process. The resulting program or procedure is called the exploit. As with any other software and procedure, the quality and reliability of the exploits varies depending on the skill and experience of the maker.

It is very important to make the distinction between a software fault, vulnerability and exploit. Several protection mechanisms [2,3,4] try to prevent functioning exploits while still at least theoretically and often also practically allowing a software bug to turn into a vulnerability. In hacker words, the bug class for the issue is still the same and valid, only the exploitation parameters changed.

## **Comparing apples with oranges for a reason**

To clarify the process of turning non-ideal software behaviour into an exploit, this article will show two examples from entirely different bug classes and explain the process of thought accordingly for each of them. Both examples are simplified fictional representations of their respective group, but the simplifications are minor and can be ignored.

The first example is a binary program called ‘uacm’ (for User-Agent Capability Matching) running on Linux. The tool’s purpose is to match the HTTP User-Agent string, usually sent by web browsers, against an ever-growing database of known web browsers and to return their capabilities, so that a web developer knows exactly which HTML version and features are going to work and which aren’t. In our example, the web developers call this command line tool from their CGI scripts on the web server and generate nice looking HTML based on its output. The tool takes the environment variable HTTP\_USER\_AGENT, provided by the web server, as input and returns the result to the standard output.

The second example is a front-end web application, called WebCRM, with an input form, requiring a username and a password for logging in as a customer. The customers cannot just register themselves; this has to be done by the company’s help desk to ensure that only valid customers are allowed. The application runs on a Windows server system and an Internet Information Server (IIS).

### ***Identification***

Every bug needs to be identified before it can become a vulnerability. The same holds true for the two examples.

The uacm binary can be tested on the Linux command line by setting the environment variable HTTP\_USER\_AGENT to arbitrary values and running the program. Alternatively, the curious hacker can also try to obtain the source code, which is unlikely, or disassemble the binary to inspect its inner workings. In this case, the hacker chooses to set the environment variable to a large string of characters, typically the character A, and runs the program.

Notice, that the hacker has completely ignored information flow of the user agent information from the web browser to the web server. He also ignored how the web server puts the information from the HTTP request into an environment variable and how the CGI might pass it on. There is a fair chance, that the attack will not work as straight forward as the test performed now, but this is not of any concern when

identifying a vulnerability in the first place. When running the program with over 200 characters, the hacker observes a crash with the message “Segmentation fault (core dumped)”. This output is the effect of a critical error occurring in the programs memory space. It could be anything at this point in time, including a faulty library or the program simply not functioning at all.

The identification of the WebCRM issue is a lot simpler. The hacker just inserts a number of non-alphanumeric characters in the username field of the application (e.g. “ ‘ % and others) and clicks on the login button. The application returns an error message, stating that the execution of a SQL statement failed due to a syntax error. The application, however, does not show the actual failing statement. When trying to login with alphanumeric characters for both username and password, the application presents a “wrong password” page.

### ***Understanding***

For the uacm program, the hacker inspects the generated memory dump in a debugger and notices a string of capital A characters has overwritten something and influenced a number of CPU registers while executing. By using ltrace [5], the hacker identifies the addresses of the last successful library calls:

```
[0x804846d] getenv("HTTP_USER_AGENT") = "AAAAAAAAAAAAAAAAAAAA"...
```

```
[0x80484a8] strcpy(0xbfb5e7a8, "AAAAAAAAAAAAAAAAAAAA"...) = 0xbfb5e7a8
```

The address of the first argument of strcpy is a stack location and inspection of the disassembled code around the caller’s address (0x80484a8) shows that the destination buffer is actually around 110 bytes, after which saved addresses of the CPU are overwritten. This is a classic stack buffer overflow scenario [2].

The WebCRM issue requires the hacker to understand what characters in username or password causes the SQL error message to be returned and what causes a “wrong password” page. Accordingly, the hacker tests each of the previous non-alphanumeric characters separate. By deducing that only the ‘ character causes an error message, he can make educated guesses about the type of issue. The character in question is widely used in SQL statements to enclose a string of data when the SQL language should not interpret it.

### ***Consequential Thinking***

At this stage, the two types of attacks on systems converge in methodology. While the hacker in case of the uacm program certainly has the advantage of running and testing the program’s behaviour in an environment controlled himself, he can certainly not assume the environment of an eventual remote target system to be the same. But while he will continue to develop an attack based on the information gathered in the local mirror-world installation, the method must work on the remote system as well to accomplish a successful attack.

The attack on the uacm program and the attack on the WebCRM system both present the same general challenge: the attacker must build a mental representation of a remote system using educated guessing and intuition. Based on this representation, which he might or might not be able to verify, he must deduce a method to influence the remote program.

Overflowing the buffer on the stack and overwriting the saved return address on the stack can probably influence the uacm program. The hacker must use his mental image to imagine the process on the remote system, anticipating issues and

inventing ways around them ahead of time. While he is able to test some of the program's behaviour on the local installation, he doesn't have any hard data on the actual conditions at the target system. The CPU architecture, operating system, web server software, environment setup and a number of other parameters influence the way the uacm process is executed. Some of these parameters can be deduced by remote identification methods known as fingerprinting, but many stay unknown. Only the assumption that an overly long string may eventually overwrite the return address of a function will hold on almost all CPU architectures. Therefore, the attack will work in principle on the target.

While the WebCRM system suffers from a completely different vulnerability, the approach of building a mental representation is the same. The special character ' is used to enclose user data in the SQL database query language. When the user data contains this character, the hacker can terminate the user data and modify the actual SQL statement. The hacker needs to make assumptions on the nature and structure of the statement he is going to modify, since it is not visible to him. His goal will be to modify the executed statement and change it's meaning in a syntactically correct way and cause the system to falsely identify him as legitimate user.

## ***Exploitation***

For the uacm example, the hacker must make assumptions on the layout of the stack on the target machine. As noted before, the hacker can overwrite critical CPU addresses on the stack of the software by supplying a long string in the user agent field of the HTTP request. Overwriting the saved return address of the affected function with an address pointing inside the buffer would cause the CPU to attempt to execute the data in the user agent string as code. Using this effect, the hacker can send custom developed machine instructions in the string instead of a series of capital A letters. If everything works well for him, the execution redirection happens, the code gets executed and he can run arbitrary functionality of his choosing on the remote system, which will typically include code to bind a listening but not authenticating shell on a port.

The WebCRM system is exploited by the attacker supplying the specially crafted string ' OR '1'='1' into the application, which will be concatenated to the SQL statement on the server side (attacker string underlined):

```
SELECT * FROM usertable WHERE username = ' OR '1'='1' AND PASSWORD=''
```

This will cause the database to return any username with an empty password. Since the developer originally planned the statement to only return a non-empty result if the username and password were known in the database, the server side code allows the login if the select statement returned at least one row. Assumed there exists at least one user without a password, the hacker got in.

## ***Reliability***

In terms of reliability, both attacks from completely different fields have the same issue of a large number of unknown factors on the target side. Therefore, the attack's implementation highly depends on the assumptions and the mental image the hacker produced. The problem is comparable to that of a software developer who has only one machine in the world to test his software. He would have to ship it to a large customer base and the customers would only be able to report back if it worked or not but no details. Part of the art form of developing exploits is to introduce dependability into something that shouldn't work at all.

## The selective fix

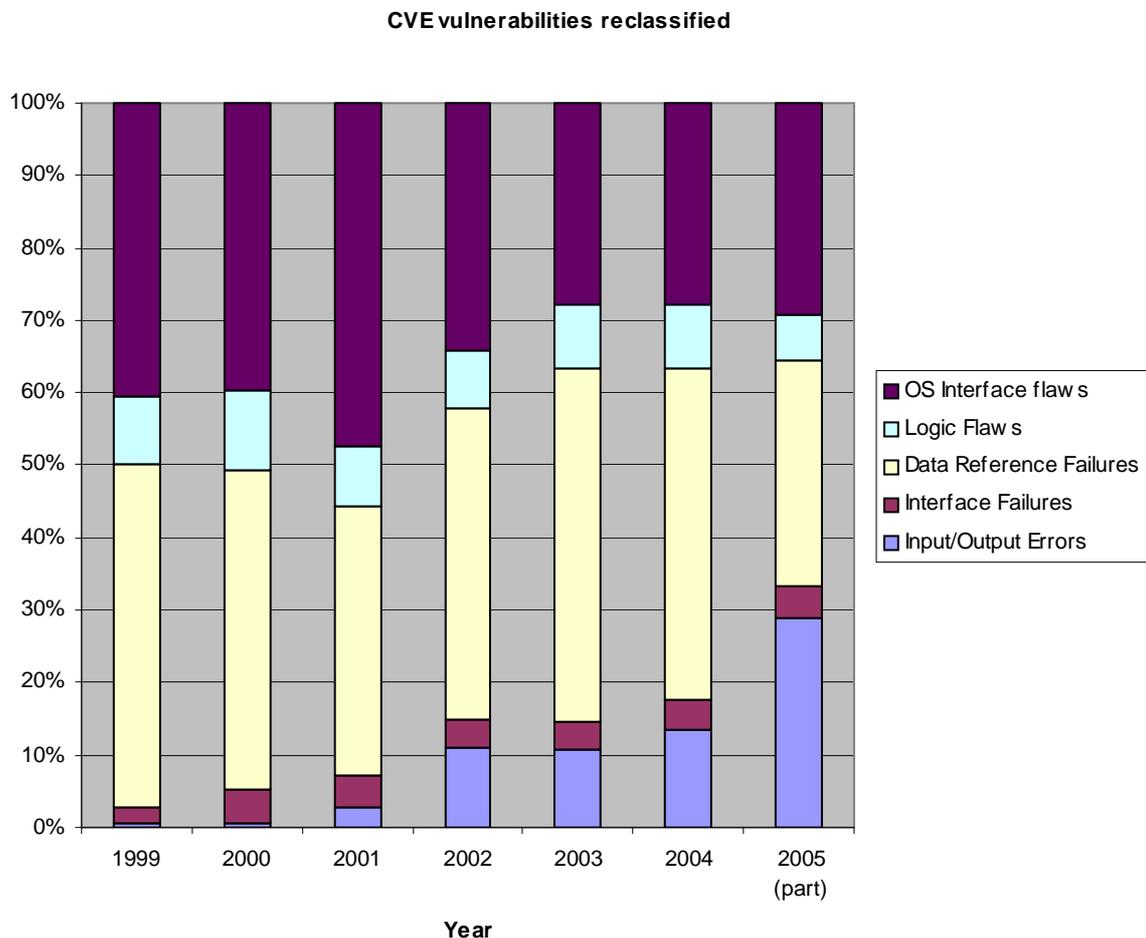
Staying with the two examples, the respective victims can fix them relatively easily. The developer of the uacm program can introduce a length check of the HTTP User-Agent string before copying it into a fixed size buffer or simply use a dynamically sized one. The developer of the WebCRM software can change the logic of his queries to the database and disallow any character other than alphanumeric in usernames and passwords.

Unfortunately, it is by now common knowledge that such selective fixes do not work very well in the long run. One bug is fixed and another is found, which can be exploited in just the same way.

## Bug Class Evolution

As mentioned in the beginning, the hacker community distinguishes between software faults by the procedure how they are exploited. This classification, however, is not very useful when trying to identify the evolution of the bug classes over several years.

While not a completely reliable and perfect data source, the Common Vulnerabilities and Exposures database (CVE) contains, as of February 6 2006 15,024 entries of publicly known security issues. I used the entire database and a very simple keyword matching script to reclassify the vulnerabilities in software fault classes known in the academic world. This remapping provides some very interesting insights in where we come from and where we go.

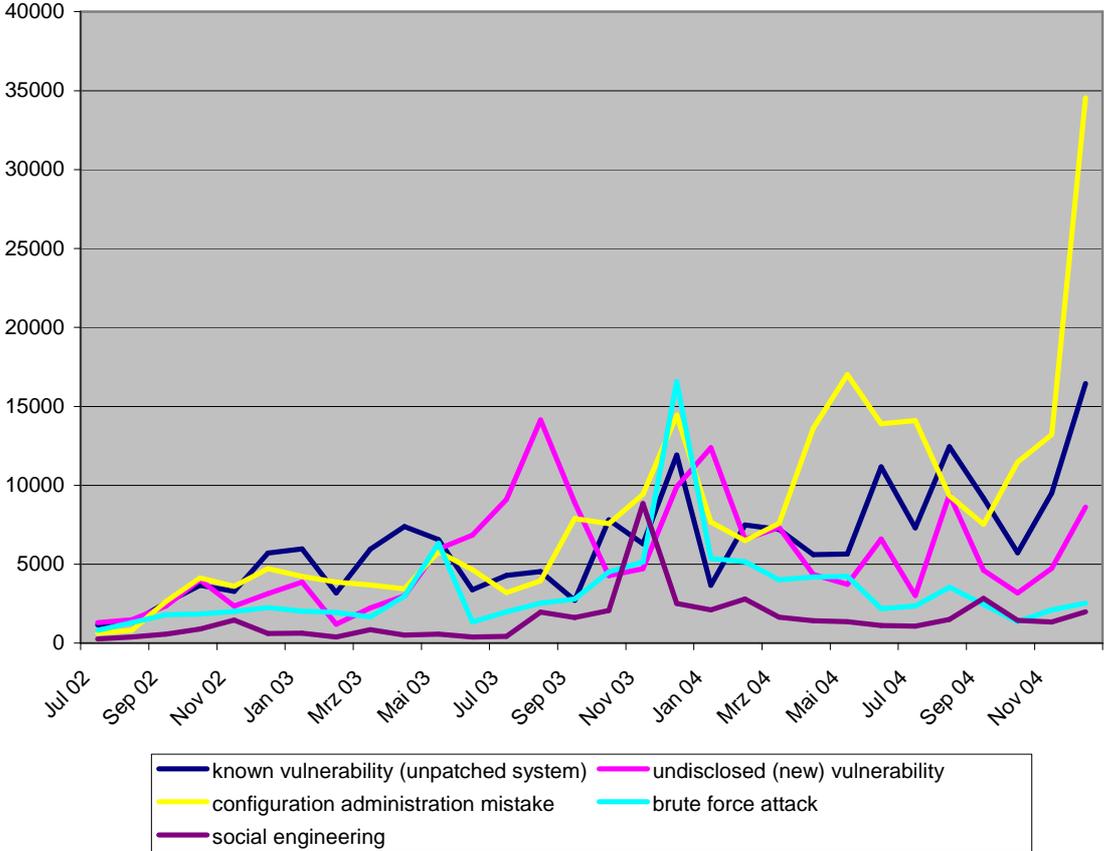


The most interesting aspect of the chart is the prominent rise of Input/Output errors. There are two main reasons for the steady increase of them since 2000. One is certainly that testing for the faults in this category (SQL injections and cross site scripting) is easier and the exploit can be made with fairly little efforts compared to the exploitation of a buffer overflow, as the comparison above showed. This attracts more people to look for these types of vulnerabilities.

The second but more important aspect of the rise of I/O errors is the change in development environments. Most people don't write critical software in C anymore, especially not in Web environments. Languages like PHP and Java are dominant in this domain. These languages are less prone to buffer overflow attacks but more to OS Interface and I/O error type of faults. It should also be noted that, probably for the same reason of more widespread use of modern programming languages, the amount of data reference faults (i.e. buffer overflows) decreases steadily over the years.

Another database supports this data from a different angle. Zone-H is an organization that, besides many other activities, lets actual attackers of web sites on the Internet submit their successful intrusions. The attacker can also select the method of intrusion used from a list. While in earlier times, many attackers used undisclosed and therefore undefended vulnerabilities to break into web sites, attacks based on configuration errors become more important these days.

Zone-H Method Chart



The same development frameworks in Java or PHP that are increasingly vulnerable to Input/Output errors are also prone to configuration mistakes due to their sheer complexity.

In general, there is a trend visible going from attacks on the most inner workings of software upwards on the abstraction level towards attacks that involve the application's own logic. In principle, these do not differ that much from a buffer overflow attack, but it is a lot easier for a hacker to change his subject from buffer overflows to SQL injections than it is to apply the defences we developed so expensively into this new area.

### ***The protection side***

In the light of this data, how did our most widely used defence mechanisms hold? In short, commonly deployed defence mechanisms such as network and application firewalls only change the picture slightly for the attacker while requiring a fairly high amount of development and maintenance. Trying to prevent discovery or a specific exploitation technique does not hold its promises when undergoing a cost/benefit analysis.

Probably the only single anti-hacker technology that has a larger impact is the introduction of the rule "write XOR execute", aimed at preventing the execution of code in memory areas that are writable. This makes sense because a legitimate reason to do so is very rare, while it's almost essential for binary exploitation.

An additional problem is the point of prevention. While, according to the ICAT database, in 2000 59% of all published vulnerabilities concerned server software, the picture is reversed in 2005 with 63% being in clients such as web browsers.

The only approach that seems to work well across the board is source code auditing, where a skilled third party re-evaluates the software design as well as the actual implementation. Unfortunately, the availability of skilled third parties is extremely limited and humans do not scale well with software doubling in size approximately every 18 months.

### **How we should go about fixing it – a cooperation proposal**

Hackers are for computer security what M Cloître and T Shinn [7] describe as *Intraspecialist Level*. Even hackers are often surprised about the turns software development and new technologies take and what impact those turns have on confidentiality, integrity and availability when seen with the eyes of a hacker.

On the other hand, computer science and especially software security and reliability research is very strong in the *Interspecialist Level* and *the Pedagogical Level*. Many of the recent breakthroughs in the hacker and computer security world were only possible because of hackers getting into or being in the academic world [6], but required them to be hackers in the first place.

The most significant point seems to be the lack of access to the other's modus of work. The type of hacker I'm referring to in this article is usually very interested in finding solutions for the challenges we as the global community of computer dependent peoples face. Many would love to work as a peer in a scientific context in a team that also has the required background of what Fleck calls "textbook science". Access to such work environments does not exist for most of them, due to the absence of academic titles, but the high demand for qualified security personal also tells them they are not worthless.

Software testing procedures and algorithms have made far too little progress since the 70s and 80s. Software security is the first time since the wide use of computers that quality is actually a real issue, since the lack of software quality is the primary

reason of insecure software. Hackers reinvent what the scientific world knows for decades, and poorly so. On the other hand, they find a lot of critical issues using their reinventions. The only intelligent solution I see is to unite.

## References

1. "Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks", Barrantes et al, <http://www.cs.unm.edu/~gbarrant/RISE.html>
2. "Detection and Preventions of Stack Buffer Overflow Attacks", Kuperman et al, ACM 0001-0782/05/1100
3. Microsoft Visual C Compiler Stack Protection, <http://msdn.microsoft.com/library/en-us/vccore/html/vclrfGSBufferSecurity.asp>
4. "GCC extension for protecting applications from stack-smashing attacks", Hiroaki Etoh, <http://www.trl.ibm.com/projects/security/ssp/>
5. "ltrace", Juan Cespedes, <http://packages.debian.org/unstable/utils/ltrace.html>
6. Halvar Flake: Structural Comparison of Executable Objects. DIMVA 2004: 161-173
7. *Expository practice: social, cognitive and epistemological linkages*, in T Shinn and R Witley, eds., *Expository Science*, Dordrecht, Reidel: 31-60 (1985)