
By FELIX “FX” LINDNER

SOFTWARE SECURITY IS SOFTWARE RELIABILITY

*Enlist hacker expertise, but stay with academic
fault naming conventions, when defending against the
risk of exploitation of vulnerabilities and intrusions.*

Recent efforts by academic researchers and the computer security industry have sought to find ways to detect and prevent software vulnerabilities from being exploited. Others have sought to find ways to detect and prevent unauthorized access to computer systems. While attack methods may differ significantly, the underlying security issues (viewed through the prism of academic software reliability research) are called “software faults.” Hackers, however, describe the same issues in different terms. Attempts to identify similarities among faults are biased toward the hacker view, as I discuss here, and often yield incomplete defenses. Missing the fact that reliability and security research addresses the same technical issues leads to inadequate approaches by the academic community [1].

RECENT BREAKTHROUGHS BY HACKERS AND COMPUTER SECURITY RESEARCHERS HAVE BEEN POSSIBLE ONLY BECAUSE HACKERS FOUND THEIR WAY INTO THE ACADEMIC WORLD YET WERE STILL REQUIRED TO BE HACKERS.

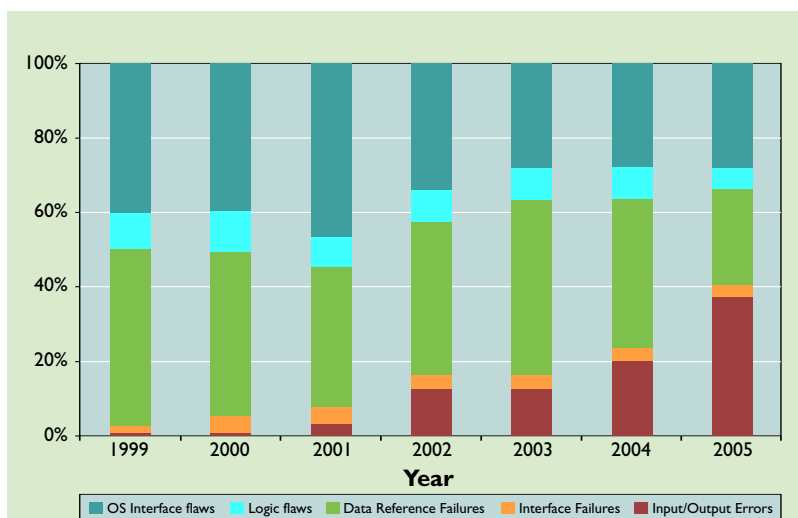
If software always worked as specified or intended by its makers, only a small subset would be vulnerable to attack, and defenses would be much easier to implement. By resolving the naming issues and looking at hard data (such as vulnerabilities and their fault classes), software defenders and users alike would be able to achieve secure, reliable software.

The science of software testing names software faults according to their root cause. Hackers name security issues according to bug class, a type of software defect they are able to exploit. When a particular type of coding fault is exploited for the first time, it becomes a bug class, and attackers search for instances of the same bug class in all other types of software, targeting similar vulnerabilities. The exploited software fault always belongs to a bug class, while most software faults don't belong to a particular bug class, since attacking hackers don't know how to exploit them.

Considering how academic researchers name software defects reveals how many obscure names from the hacker community emerge; for example, in software testing, buffer overflows and integer overflows are called "data reference failures." Most so-called denial-of-service problems in software are data reference failures, some falsely classified as "denial of service," rather than "nonexploitable buffer overflows." This leads to the perception that the fault cannot be used to break into the system, though only the person who classified it is unable to break in. This nomenclature is evidence that naming is based on exploitability rather than cause and thus often produces solutions that ignore entire groups of faults.

So-called format string bugs and several types of

race conditions belong to the fault class known as "interface failures." Directory traversal bugs, illegal directory or file access, and remote command execution almost always turn out to be interface failures. The difference is that the interface failing is so important that these faults deserve their own name—"operating system interface failures."



Common vulnerabilities and exposures reclassified using terms from software reliability research.

Two bug classes that have emerged in the context of Web applications are "SQL injections" and "cross-site scripting vulnerabilities," each of which can be classified as input/output errors. Academic research in reliable software and software testing has identified several other defect types that hackers have not (yet) been able to exploit and hence have not yet named.

How do hackers distinguish between a software bug that is not security-related and a vulnerability? For example, finding all potential buffer overflows in a given piece of software may yield a number of findings; none, some, or all might be classified as a vulnerability. Classification depends on who causes the condition, what is the condition's most severe effect, and how the condition might be used to perform an

action that would normally be restricted. If the *who* question can be answered with “hacker,” the *what* question with an (at least) “mostly controllable condition,” and the *how* question with “a known procedure for this bug class,” hackers call it a vulnerability, since they are able to exploit it. A piece of software written to automatically take advantage of the situation is called “the exploit.”

Developers must distinguish among software fault, vulnerability, and exploit to be able to build effective defenses. Protection mechanisms [3, 5, 6] aim to prevent functioning exploits while at least theoretically (and often practically) still allowing a software bug to turn into a vulnerability. In hacker parlance, the bug class is the same and valid; only the exploitation parameters have changed.

To clarify the process of turning nonideal software behavior into an exploit, I offer two simplified, fictionalized examples from entirely different bug classes and explain the thought processes needed to exploit each of them. One is a binary program called “user-agent capability matching,” or uacm, running on Linux. The tool matches the HTTP user-agent string, usually sent by Web browsers, against a growing database of known Web browsers and identifies their capabilities; this way, a Web developer knows exactly which HTML version and features will work and which won’t. Web developers call this command line tool from their common gateway interface scripts on a Web server to generate elegant and functioning HTML based on its output. The tool takes the environment variable `HTTP_USER_AGENT` as input and returns the result to the standard output.

The second example is a front-end Web application called “Web customer relationship management,” or WebCRM, involving an input form requiring a username and a password for logging-in as a customer. In this application, customers cannot simply register themselves.

As I discussed earlier, a bug must be identified by the hacker before it becomes a vulnerability. The same must happen in 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 then running the program. In this case, the hacker chooses to set the environment variable to a long string of characters (typically the character A) and runs the program. There is a fair chance that the attack will not be as straightforward as the test, but this is of no concern to the hacker when identifying a vulnerability.

When running the uacm binary with more than 200 characters, a hacker would observe a crash that produces the message “segmentation fault (core

dumped)” as a result of a critical error occurring in the program’s memory space. Identifying a potential issue in WebCRM is different. The hacker inserts a number of nonalphanumeric characters (such as “,” “’,” and “%”) in an application’s username field and clicks the login button. The application returns an error message stating that the execution of a SQL statement failed due to a syntax error, though the application does not show the failing statement. When trying to login with alphanumeric characters for both username and password, the application presents a “wrong password” Web page.

For the uacm binary, the hacker identifies the addresses of the last successful library calls by using ltrace, which is designed to record calls to library functions and their arguments [2], as in this classic stack buffer overflow scenario [5]:

```
[0x804846d] getenv("HTTP_USER_AGENT")
= "AAAAAAAAAAAAAAAAAAAA" ...
[0x80484a8] strcpy(0xbfb5e7a8,
"AAAAAAAAAAAAAAAAAAAA"...) = 0xbfb5e7a8
```

Here, the address of the first argument of `strcpy` is a stack location. Inspection of the disassembled code around the caller’s address [0x80484a8] then shows that the destination buffer is approximately 110B, after which saved addresses of the CPU are overwritten. For the WebCRM application, a hacker would test each of the previous nonalphanumeric characters separately. By deducing that only the ‘ character causes an error message, the hacker would then make an educated guess about the type of issue—in this case a SQL injection.

The methodologies of the two types of attacks converge at this stage in the process of identifying a vulnerability and turning it into an exploit. The attack on the uacm binary and the attack on the WebCRM application each present the same general challenge. The attacker must build a mental representation of a remote system through educated guessing and intuition. Based on this representation, which a hacker might or might not be able to verify, the hacker would have to deduce a method to influence the remote program. The hacker imagines how the process operates on the remote system when overflowing the buffer on the stack and overwriting the saved return address on the stack to control the remote program. While the WebCRM application suffers from a completely different type of vulnerability, the steps the hacker must take in building a mental representation are the same

THE MOST PRESSING ISSUE IN THE COOPERATION BETWEEN HACKERS AND ACADEMIC RESEARCH IS A LACK OF ACCESS TO EACH OTHER'S WORK METHODS.

as with the uacm binary. When the user data contains the ' character, the hacker is able to terminate the data and modify the actual SQL statement. The hacker must also make assumptions concerning the nature and structure of the statement to be modified, since it is not visible. The aim is to modify the executed statement and change its meaning in a syntactically correct way, causing the system to falsely identify the hacker as a legitimate user.

In the uacm binary example, the hacker must make assumptions concerning the layout of the stack on the target machine. Overwriting the saved return address of the affected function with an address pointing inside the buffer would cause the CPU to attempt to execute (as code) the data in the user agent string. Exploiting 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, execution redirection occurs, the code is executed, and the hacker can run arbitrary functionality of the hacker's choosing on the remote system.

The WebCRM application is exploited by supplying the specially crafted string `` OR `1'='1'` as the username without a password; it is then concatenated to the SQL statement on the server-side (attacker string underlined):

```
SELECT * FROM usertable WHERE  
username = ` OR `1'='1' AND PASS-  
WORD=' '
```

The string causes the database to return any username with an empty password. If at least one user exists without a password, the hacker gets in.

The respective victims are, however, able to fix the bugs relatively easily. For example, 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. And the developer of the WebCRM application software can disallow any character other than alphanumeric ones in usernames and passwords. Unfortunately, it is common knowledge that such selective fixes do not work well in the long run. Where there is one bug, there are others, essentially

representing a quality problem in the software.

BUG CLASS EVOLUTION

While not a perfect data source, the Common Vulnerabilities and Exposures database (cve.mitre.org) contains (as of Feb. 6, 2006) 15,024 entries of publicly known security issues. I used the entire database and a simple keyword-matching script to reclassify the vulnerabilities from hacker terms into a number of the software fault classes known by academic researchers.

This remapping yielded several interesting insights concerning the evolution of bug classes (see the figure here). The most interesting is the prominent increase of input/output errors since 2000, likely occurring for two main reasons: ease of testing for faults and a gradual change in development environments. For example, it is easier to test for faults in the category of SQL injections and cross-site scripting. Moreover, the number of Web-based systems has also increased, along with the number of potential targets. More and more potential targets attract more and more hackers to look for this type of vulnerability.

Meanwhile, most programmers don't write critical software in C anymore, especially in Web environments, (the most widely attacked systems), due to their visibility and Internet-wide accessibility. Dominant in this domain are languages like PHP and Java that are less prone to buffer overflow attacks but are more likely to produce operating system interface- and input/output-error-type faults. Also, along with increased use of modern programming languages has come a steady decrease in the number of data-reference faults, or buffer overflows over time, due to their built-in boundary checks.

An emerging trend among hackers involves going from attacks on the deepest level of software upward on the abstraction level toward attacks involving the application's own logic. In principle, these attacks do not differ much from buffer overflow attacks, but it's much easier for attackers to adjust their methods

from buffer overflows to SQL injections than to modify specialized defenses to prevent new attacks.

An additional problem is identifying the point of prevention. The National Vulnerability Database [7] reported that in 2000, 59% of all published vulnerabilities concerned server software. The picture had reversed in 2005 with 63% of all published vulnerabilities concerning clients (such as Web browsers). This finding does not mean that servers are more secure but that attackers moved on because there is still plenty of vulnerable software around.

Commonly deployed defense mechanisms (such as network and application firewalls) have changed the picture only slightly for attackers while requiring a significant amount of additional development and maintenance work on the part of system administrators. Ways to prevent specific exploitation techniques are likely to be rejected when undergoing thorough cost/benefit analysis.

The anti-hacker technology that probably has the greatest impact on system security is the rule write XOR execute, which declares memory either writable or executable, never both. It is aimed at preventing the execution of code in memory areas that are writable but does not affect legitimate applications and defies exploitation. The rule was developed by hackers who gave it to both the software development and hardware industries.

The only approach that seems to work well for identifying vulnerabilities and protecting all kinds of systems is source code auditing, whereby a skilled third party reevaluates the software design and its implementation. Unfortunately, the availability of skilled third parties is limited, especially with the amount of software worldwide doubling in size approximately every 18 months.

Hackers are for computer security what [8] described as the “intraspecialist level,” or highly motivated experts in their own limited field. Software security and reliability research is very strong at both the interspecialist level and at the pedagogical level and is found in the academic community and in authoring textbooks. Recent breakthroughs (such as binary code-matching algorithms) by hackers and computer security researchers have been possible only because hackers found their way into the academic world [4] yet were still required to be hackers.

The most pressing issue in the cooperation between hackers and academic research is a lack of access to each other’s work methods. The more accomplished hackers are motivated to find solutions for the security challenges faced by the worldwide

community of computer-dependent people. They would thus work well as peers in a scientific context. What they need is a team of researchers with the required background, or what Ludwik Fleck, a Polish biologist, calls “textbook science” [9]. Access to such work environments is not available for most hackers due to their lack of academic titles, credentials, and status, but strong demand for qualified security experts signals the value of their expertise.

CONCLUSION

Software testing procedures and algorithms have advanced only incrementally since the 1970s and 1980s. Software security is the driving force behind the need for software quality, since a lack of quality is the primary reason for insecure software. Hackers have managed to reinvent what the world of computer science has known for decades, only poorly. On the other hand, they identify a lot of software vulnerabilities through their innovation. The only solution for making systems as secure as we need them to be by eliminating software vulnerabilities is for hackers and academic researchers to unite. ■

REFERENCES

1. Barrantes, E., Ackley, D., Forrest, S., Palmer, T., Stefanopvic, D., and Zovi, D. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security* (2003); www.cs.unm.edu/~gbarrant/RISE.html.
2. Cespedes, J. ltrace. Online documentation; packages.debian.org/unstable/utils/ltrace.html.
3. Etoh, H. *GCC Extension for Protecting Applications from Stack-smashing Attacks*. Technical report and source code, first published May 8, 2001; www.ttl.ibm.com/projects/security/ssp/.
4. Flake, H. Structural comparison of executable objects. In *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment* (Dortmund, Germany, July 6–7, 2004), 161–173.
5. Kuperman, B., Brodley, C., Ozdoganoglu, H., Vijaykumar, T., and Jalote, A. Detection and prevention of stack buffer overflow attacks. *Commun. ACM* 11, 48 (Nov. 2005), 50–56.
6. Microsoft. *Visual C Compiler Stack Protection*. Microsoft Visual Studio 2005 documentation; msdn.microsoft.com/library/en-us/vccore/html/vclrfGSBufferSecurity.asp.
7. National Institute of Standards and Technology. National Vulnerability Database, Gaithersburg, MD; nvd.nist.gov/.
8. Reidel, D. Expository practice: Social, cognitive and epistemological linkages. In *Expository Science*, T. Shinn and R. Witely, Eds., 1985, 31–60.
9. Trenn, T. and Merton, R., Eds. *The Genesis and Development of a Scientific Fact*. University of Chicago Press, Chicago, 1979.

FELIX “FX” LINDNER (fx@sabre-labs.com) runs SABRE Labs, a computer security consulting company in Berlin, Germany.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
