Increasing system security by interdomain communication analysis and brute-force auditing

Wojciech A. Koszek
dunstan@FreeBSD.czest.pl
IX Liceum Ogólnokształcące im. C. K. Norwida w Częstochowie
Krajowy Fundusz na Rzecz Dzieci

Abstract

Typical approach to communication in modern computer environments is based on passing data formated with strictly defined rules between two or more end-points. Detailed analysis of such process may lead to interesting results, especially in computer security area. This paper is a result of research related to ways of communication between untrusted domains in UNIX operating system and a new methodology of source code auditing. Author has considered not only typical and well known technical problems, but also man-hour cost and target of auditing strategy.

1 Introduction

Auditing has always been hard for software projects, especially when such software is critical for data and machine security. Almost every layer of software structures might be a potential danger: from operating system, through user applications (text editors, window managers) to remote access services (OpenSSH *sshd* daemon). Practise has shown that auditing is costly in the sense of time and the work which needs to be spent in order to bring useful results. In this paper author presents effecitive strategy of auditing, which divides software product into untrusted domains, and shows potential vectors of attack which can be used by a hacker. By analysis of such a relation between domains, sometimes very specific problems can be discovered easily. Presented methodology of exploitation, later called brute-force auditing, can be used to mass-disclosure of dangerous, often critical problems.

This paper is divided into four parts. Second chapter is a general introduction to the topic: presents audit strategy, explains nomenclature used (domain definition, domain type and characteristics) and communication ways discussed. Third chapter contains results of author's work and experiences gained from UNIX code auditing, most common and possible sources of existing problems. Fourth chapter contains summary information, effects and future directions of the research.

Sample bugs presented in this paper were discovered in the FreeBSD system [1], derived from BSD, the version of UNIX developed at the University of California, Berkeley. Thus every case refers to the FreeBSD environment. Since mechanisms available in that system are widely used in other UNIX variants, results of research may concern also other systems, which won't be discussed.

2 Strategy of audit

In modern computer environments communication between domains is present everywhere. A typical user does not even imagine how much time and power is spent on transferring data used for further processing. Although "domain" has many meanings,

this paper uses only one of them – a domain as untrusted, separated environment, which acts as a potential sender/receiver of data packed in defined format. Thus, a domain might be understood as a function, complete application, operating system kernel, networked host or group of hosts. These completly different objects have common characteristics: all of them require data packed in format consistent for both communication participants. Also, the most common case is dependency of later processing on result obtained earlier. This is why corruption of single object very often causes undefined behaviour of entire domain.

Of course context will change: while speaking about low-level function written in machine code, communication will be based on pushing arguments in a correct order onto the stack, popping result from the stack (architecture-specific operations), but while transferring data between kernel and user, communication will be based on allocating data in higher layer. Such a situation looked interesting, since not always both sides of communication exchange data in consistent way, not always programmer is able to predict format of input data. Simply not every domain may be called "trusted". This is why audit might be more effective if done in specified order:

- cross-domain communication analysis
- data integrity checks
- validity checking
- implementation details

Although these conditions look obvious in theory, they are all intertwined with each other in practice, this is why author dropped such division just at the beginning of the research.

In the closed source software, analysis of these cases pushes researchers to advanced reverse engineering techniques [2], [3]. In the open source software this is possible by source audit. Auditing of every source file in an operating system distribution would not be possible for one person and would be very difficult for a team of researchers. Full audit was firstly chosen by OpenBSD developers [4], but amount of work caused that this idea was dropped. In order to chose proper vectors of attack, question to be asked before starting research was – "What to audit?". Factors which directed author's eyes to specific areas of operating system functionality were:

- availability for the user
- possibility of exploitation
- consequences of eventual attack

It was obvious that availability will probably mean system applications auditing. Only those with privileges elevated in execution time were chosen. Possiblity of exploitation differs between attacks researched and is highly dependant on auditors' knowledge, thus bugs which fistly looked complicated, at the end of the research seem to be obvious. The results are most interesting from the attackers' point of view. Not only machine overtake were studied, since these attacks appear to be less popular, but mostly Denial of Service (*DoS*) attacks, which in many cases may be more dangerous: Internet hosting

and scientific environments where continuous computation, access and availability are needed.

Order of audit and chosen vector of potential attack need to be analyzed separately. Control flow is completely different in user application and in the internal system structures. These situations have shown the need of automatization of audit process. Of course, there are some commercial products, which can help open source projects [5], but none of them is publicly available, making it at once hard to extend, test and prove that such testing software does not bring false sense of security. Mass-auditing could raise software security several times. At the first glance, this might not seem as an effective way of analysis. In fact, the truth is completely different: automated and well planed testing strategy was the reason for discovery of the bugs presented in this paper with minor man-hour cost.

For the sake of the discussion on the research results communication might be divided in three classes:

Function to function This topic will not be discussed here in great detail since it has become very popular ([6],[7],[8]) and is well known nowadays. Basic situation: implementation of machine instruction interpretation lets the attacker modify normal instruction flow by injection of malicious code onto the program's stack which executes it automatically, the same doing attacker bidding.

Operating system to application in user space Typically not taken into account, but present in UNIX systems family from the beginning. Examples might be well known functions responsible for system environment interaction – *sysctl(3)* or kernel notifications mechanisms like *devfs* and *procfs* filesystems.

User-space application to kernel-space: The most dangerous, since successful attack may lead to full access to target machine or at least cause denial of service and crash.

Also methodology of attack differs between problems discovered: audit of source code block chosen by untrusted domain analysis and so-called brute-force methodology. Domain analysis has been discussed above. Brute-force methodology is not very popular but very effective: even without very detailed knowledge, a determined attacker can cause system crash. Reason is simple: brute-force auditing is based on passing random data, but doing it dozens or hundreds of times, while observing application behaviour.

3 Results

Typical way of doing audit is to review program's source code line by line with security kept in mind. This way of security review is most effective, but good for small, non-complicated programs. If application grows in time, audit will become difficult, is only possible. This chapter presents alternative methods of auditing, which have been used by the author.

3.1 Interdomain communication analysis

ioctl(2) function was the main subject of cross-domain communication study, but also other ware inspected as well. Context differs between functions (**cntl* functions require file descriptor as their basic argument, were either *sysctl(3)* operate as is, in a thread context or eventual arguments need to be packed respectively), but the general scenario remains the same: data packed in user-space gets transferred from code executed in the least privileged procesor level (so called rings on x86 architecture) to kernel space. This address space is directly available for code executed in the most privileged level - operating system kernel. Thus, kernel needs to make proper interpretation: copied data might be the content which is ready for being dispatched, or it may contain pointers to additional data. First situation is simplier, and does not need explanation. Second case is more complicated, since it requires from kernel programmer to copy data located in user fragment of address space to buffer with enough space to contain data structure. This operation typically is not dangerous, since tools available with system distribution are used for proper interpretation of structures' fields, thus filling them with proper data. This is why *ifconfig(8)* will not put negative value in the '*len*' – field of `ifconf` structure describing length of network interface list available in system. But inquisitive user may choose that value, and discover new ways of system exploitation. This is a first example of a discovered bug [13]: *ifconf()* did not do data validity checking passed and thus used negative value to allocate memory through *malloc(9)*.

```
 1 panic: wrote past end of sbuf (0 >= 0)
 2 KDB: stack backtrace:
 3 panic(c0663212,0,0,d94b0bd8,c04f6a8f) at panic+0xeb
 4 _assert_sbuf_integrity(c1a2a4a0,c1a2a400,c1bfb2e0,d94b0c34,c053dc99) at _assert_
 5 sbuf_integrity+0x3b
 6 sbuf_bcat(c1bfb2e0,d94b0c08,10,1,0) at sbuf_bcat+0x1b
 7 ifconf(c1c86dec,c0086924,d94b0c60,c1bd1780,c1bd1780) at ifconf+0x129
 8 ioctl(c1bd1780,d94b0d14,3,0,292) at ioctl+0x11e
 9 syscall(2f,2f,2f,0,bfbfecc8) at syscall+0x128
10 Xint0x80_syscall() at Xint0x80_syscall+0x1f
11 --- syscall (54, FreeBSD ELF32, ioctl), eip = 0x8048517, esp = 0xbfbfe81c, ebp =
12  0xbfbfec78 ---
13 Uptime: 4m38s
```

Lines 10-7 show instruction flow from trap gate handler (also know as "software interrupt"), system call and *ioctl(2)* handlers, and finally *ifconf()* function. Later lines show function calls, where crash has happened.

Similar problem existed in *if_clone_list()* function [14]: passing negative values via *if_clone* structure lead to OS crash (also known as kernel panic).

Analysis of TTY size handling code seemed to be interesting. Since setting virtual terminal size is obligatory action taken by the user, user-space application available with standard software distribution, *stty(1)*, could be used. Typical values were replaced with negative ones, which lead to a disclosure of bug [17] in *editline(3)* library functions responsible for memory allocation. Important applications were linked with this library, from which *cdcontrol(1)*, *lpc(8)*, *pppctl(8)* were executed with elevated privileges.

These three examples presented above were caught by auditor's eyes. Audit took time, and although results were important for system developers, it was also proof that some amount of time counted in man-hour needs to be sacrificed, in order to raise security level. It also showed that keeping functional programming secure not always is a major target even for very experienced and well educated programmers.

3.2 Brute-force auditing

Bugs in *lpc(8)* [15], which is funny, in *rs(1)* [18] and compability code responsible for handling `PECOFF` binaries [16], were disclosed with brute-force method. First case was present in *lpc(8)* application. Linked with *editline(3)* library (handling for a used input), is a line printer control program. If data came from a terminal, it used *el_gets(3)* function, otherwise, *fgets(3)* was used. User could send malicious data through *fgets(3)*, skipping variable initialization for *editline(3)* library, and causing *lpc(8)* to crash.

rs(1) used for reshaping data array took number of rows and columns from command line. Due the lack of validity checking, it had problems with handling malicious values. Trivial script written in Perl passed string, group of strings and a number as command line arguments to applications in random order. After *rs(1)* received `SIGSEGV`, it was easy to track:

```
$ echo test | rs 1 -99999999999
zsh: done                                echo test |
zsh: segmentation fault (core dumped)  rs 1 -99999999999
```

Handling of executable files may also be used as communication way, since low level structures of an executable are interpreted by operating system kernel in order to properly initialize structures needed for further execution. The idea of this attack was to provide malformed content and cause system crash. Objective was reached and proved not only direct danger, but also low level of difficulty of such an attack: to satisfy validity and integrity checks first few bytes were taken from normal executable file, the rest was filled with data comming from */dev/urandom*. Everything was done by the */bin/sh* script which looped actions described above and worked for about 10 minutes. Method might be called semi brute-force auditing, since data was not fully random.

3.3 Conclusion

Disclosure of these problems were coordinated with FreeBSD team – either through Security Officer or FreeBSD mailing lists. These local DoS attacks were reported and fixed, just like other problems, which had to be coordinated (*editline(3)* library code is shared and synchronized with NetBSD distribution).

4 Summary

User-space application analysis did not differ from a typical situation: after obtaining memory dump file, gdb could be used to analyze it, thus documenting a bug in a matter of minutes. Author has found the UNIX kernel auditing topic most interesting. Differences between user-space and kernel-space seemed to be the biggest problem, and analysis took most of the time, since it requires deep understanding of almost every aspect of system architecture: chosen coding conventions, data structures, function flow and methods of data dispatching ([9], [10], [11], [12]). Author has to admit that missed one problem related to static buffer allocation in *ifconf()*. The bug was discovered few days after author's local DoS disclosure and was based on fact, that space for kernel data structures is allocated from one piece of address space managed by kernel allocator, which does not clear memory after object releasing. Memory area destined for object targeted to user-space needs to

be zeroed explicitly with *bzero(3)* or `M_ZERO` in *malloc(9)*, so that no additional data will leak after data movement from kernel to user space. It also seems that kernel-side buffer overflows are not very common these days, since none of almost 4000 *ioctl(2)* handlers appeared to be vulnerable to that kind of attack. Although kernel-related problems seem to be the most time-consuming, results should satisfy potential attackers. They are also self-documenting – discovery of bug makes it unnessesery to analyze with kernel debugger. Author helped to increase FreeBSD kernel security and to develop a few fixes for kernel-side code. Although programmers are focused on secure solutions while bringing new features to operating systems, there are still some undiscovered bugs.

References

[1] FreeBSD project, *http://www.FreeBSD.org*.

[2] Working Conference on Reverse Engineering *http://www.cc.gatech.edu/conferences/*, (1995-2005).

[3] "Buffer Overrun in Microsoft RPC service", LSD Research Group, 2003 *http://www.lsd-pl.net/special.html* .

[4] OpenBSD Project, *http://www.OpenBSD.org*.

[5] "Automated Error Prevention and Source Code Analysis" – *http://www.coverity.com/* .

[6] "Smashing The Stack For Fun And Profit", Phrack 49, AlephOne, 2000.

[7] "UNIX Assembly Codes Development for Vulnerabilities Illustration Purposes", LSD Research Group, 2002 *http://www.lsd-pl.net/unix_assembly.html*.

[8] "Kernel Level Vulnerabilities – Behind the Scenes of 5th Argus Hacking Challenge", LSD Research Group, 2002 *http://www.lsd-pl.net/kernel_vulnerabilities.html* .

[9] "The Design and Implementation of the 4.4BSD Operating System", Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, John S. Quarterman, 1996.

[10] "UNIX internals – The New Frontiers", Uresh Vahalia, 1996.

[11] "The Design and Implementation of the FreeBSD Operating System", Marshall Kirk McKusick, George Neville-Neil, 2005.

[12] "Solaris Internals", Richard McDougall and Jim Mauro's, 2000.

[13] "Local DoS from user-space in ifconf()", W.A. Koszek, 2004 http://www.freebsd.org/cgi/query-pr.cgi?pr=kern/77424.

[14] "Local DoS from user-space in *if_clone_list()*", W.A. Koszek, 2005 *http://www.freebsd.org/cgi/query-pr.cgi?pr=kern/77748*.

[15] "Use of uninitialized variables in lpc(8)", W.A. Koszek, 2004 *http://www.freebsd.org/cgi/query-pr.cgi?pr=bin/77462*.

[16] "Local DoS in sys/compat/pecoff (+ other fixes)", W.A. Koszek, 2005 *http://www.freebsd.org/cgi/query-pr.cgi?pr=kern/80742*.

[17] "Misuse of *el_init()* can lead multiple programs to SIGSEGV", W.A. Koszek, 2005 *http://www.freebsd.org/cgi/query-pr.cgi?pr=bin/80346*.

[18] "rs(1) handles command line arguments improperly (SIGSEGV)", W.A. Koszek, 2005 *http://www.freebsd.org/cgi/query-pr.cgi?pr=bin/80348* .