Buffer overflows on linux-x86-64

Hagen Fritsch <fritsch+stacksmashing@in.tum.de> Technische Universität München

January 22nd, 2009

Revision 2. For BlackHat Europe, April 16th, 2009

Contents

1 Introduction

Buffer overflows are the most serious, most everlasting, most impactous and one of the most well researched software vulnerabilities. With the first exploited buffer overflow being dated back to 1988, two decades later the problems are still prevalent. Although several techniques to minimize exploitation success have been developed and are deployed in modern operating systems, buffer overflows still exist today and sophisticated exploitation techniques continue to allow attackers to break into systems and to execute arbitrary code.

Since the public knowledge even among security-aware people is rather limited to the classic buffer overflows as described by Levy (1996) and the well-known literature does not reflect the current state of the art, this paper is going to give an overview on modern mitigation techniques, their employment and their weaknesses providing the reader with enough knowledge to assess the threat of buffer overflows as of today.

2 Mitigation techniques

This paper is not going to explain the basic idea of buffer overflows again and assumes everyone to have read and understood the Levy (1996) article that appeared in Phrack magazine issue 49 and that everyone cites.

As the problem of buffer overflows has been know for a long time, mitigation techniques hindering buffer overflows from being exploited were developed. There are two different approaches. The first is to make software safe, by verifying code and ensuring that there cannot be any buffer overflows (cf. Section 2.1), the other approach tries to reduce the likelihood of exploitation. For the latter category there are three techniques which are widely deployed:

Non-executable stack, heap, data sections. As classic buffer overflows rely on the injection of arbitrary code and executing it, preventing applications from executing code on writeable pages stops this form of operation as section 2.2 is going to discuss. Several techniques such as the *return-into-libc* measure (cf. Section 2.2) allow still for arbitrary code execution.

Address Space Layout Randomization (ASLR). Classic buffer overflows and methods working around non-executable stacks heavily rely on known fixed addresses, which ASLR addresses by randomizing the addresses of certain pages in the process' address space (cf. Section 2.3). A collection of techniques working around this problem has been developed.

Stack Smashing Protection (SSP). Since the heart of most buffer overflows lies in overwriting a return address on the stack to redirect the execution flow, several sorts of protection and detection measures have been developed which Section 2.4 is going to discuss.

2.1 Buffer Overflow Prevention

While the standard techniques described in this section try to mitigate the impact of buffer overflows after they happened, the most obvious approach would be to prevent buffer overflows from occuring in the first place. The core problem is pointer arithmetic being used in programming languages like C or C++. The language itself is safe if used appropriately, but C makes it very easy to make subtle errors that result in buffer overflow vulnerability issues (see the chapter "C language issues" in Dowd et al., 2006). Modern high-level languages such as Java or Python are strongly typed and therefore buffer overflows cannot occur by design (Cowan et al., 2000) (unless there are implementation bugs in the language, but this is not to be considered here). However, most applications are still written in C, thus it remains to be questioned if there are ways to make C safer. The summary of Cowan et al. mentions a **bounds checking gcc** extension, which checks that buffer-accesses remain in bounds. This can be formally proven (Gough, 2004) and allows to run verified secure (in terms of buffer overflows) software. Unfortunately, there is no packaged version of this extension available, thus the majority of developers will not be likely to use it. Additionally and this is even more important Gough mentions that the bounds-checking introduces a lot of overhead code, since pointers and buffers are accessed frequently. Experiments showed a slowdown of 10 times or more.

Static verification approaches can find possible vulnerabilities to some extend, but fail for complex program structures and are very hard to implement (Chander et al., 2007).

For **dynamic analysis** there is also `valgrind`, a sophisticated tool mainly used during the debugging process working on raw binaries used to discover memory leaks and illegal memory accesses. However it still does not guarantee that all possible code-paths are followed and buffer overflows within a program's structure can still occur. Several approaches such as the one of Chander et al. try to combine static and dynamic analysis of software to mitigate the runtime problem at least partially.

Unfortunately, most of these approaches are purely academic and are not yet helping to avoid buffer overflows. This is why the current techniques deployed in operating systems aim on reducing the possible impact buffer overflows can have. Some of these techniques have the advantage, that they do not require modification / recompilation of existing software, but can be applied on an operating system level securing all running applications.

2.2 NX — Non-executable data-pages

Buffer overflows were (and are still) typically exploited by supplying code to the application to which the control-flow will be redirected after the buffer overflow led to some corruption in memory. This code is typically posted in the heap or directly on the stack, sometimes in arguments or environment variables (which reside on the stack as well). To prevent the execution of attacker supplied code all data pages of a process (esp. the stack and the heap) are marked as non-executable. Additionally a process' code pages are not writeable, leaving an attacker no way to insert own or modify existing code. This technique is also known as *Data Execution Prevention* (DEP). There are two issues concerning this protection:

Backward compatibility: A lot of software relied on being able to modify its code or used dynamically constructed trampoline functions to achieve different tasks. Therefore several applications explicitly modify the mappings to allow such functionality, thus opening a door to an attacker.

Hardware support: While all modern x86 processors support page-tables with the NX bit allowing to specify if a page is executable or not, the traditional x86 architecture allowed such protections only in conjunction with segmentation. But since most operating systems use page tables instead of segmentation, IA-32 processors manufactured before 2004 do not include this functionality. However, it is possible to still enforce such a protection using software emulation like the PaX-project successfully demonstrated.

Return into libc

With the NX-bit set on the stack, the execution of own code is not possible any more. But attackers came up with other ways of executing code. The basic idea is, that many libraries are usually mapped into a process' address space. These libraries already contain most of the code an attacker might want to execute. A typical example is the execution of the `system(3)` function to spawn a shell, so that the attacker can execute arbitrary commands. Since the `system(3)` function is part of the C-library, the attacker does not need to write and inject his own code, but can just go ahead and use the already existing functionality. For buffer overflows, the attacker usually controls the stack and as such parameters for the functions can be set up accordingly, as calling conventions pass arguments on the stack. These calling conventions changed however for x86-64 and parameters now have to be passed in registers. Nergal (2001) summarized some advanced return-into-libc techniques, which include finding instructions in a library that will pop a register off the stack and return to the next return address on the stack afterwards. Such assembly code might look like this: `pop %rdi; ret`. If an attacker finds such an instruction, he or she can set up a stack frame such that the RDI register is popped off the stack and the `ret` continues by calling a library function now with the argument in the right place.

This technique can be extended to collect arbitrary instructions followed by `ret` statements and then chunk those together so that arbitrary combinations of instructions become possible again. Another well-liked technique is to use a function call to `mprotect(2)` to mark stack or heap pages as executable again and then jumping there to the attacker-supplied code.

2.3 Address Space Layout Randomization

To be able to reliably exploit vulnerabilities, attackers need to know exact addresses and offsets. In the past attackers were already able to exploit buffer overflows even if they did not know the *exact* address of where their buffers are located, because this depends on a lot of factors, esp. environment variables, software version and compiler used. The attackers' work-around was to supply large amounts of `nop` instructions, because the differences were usually in the range of less than a kb. Address Space Layout Randomization (ASLR) is an explicit measure to harden the system against such attacks rendering most exploits that rely on known fixed address unusable. While traditionally libraries, stack and heap were assigned to fixed addresses within an executable, with ASLR these are randomized in varying degrees. In the virtual address space, each page has its virtual address which references some real memory via page tables. ASLR

```
| virtual address  |  offset   |     | (sign extension) | virtual address |  offset   |
|    (20 bits)     | (12 bits) |     |    (16 bits)     |    (36 bits)    | (12 bits) |
```

Figure 1: Virtual addresses on i386 and x86-64

randomizes some bits of the virtual address. This could theoretically be up to 20 bits (or 36 bits on x86-64), but in practice the amount is limited due to constraints imposed by the operating system. Since kernel 2.6.12 ASLR is enabled by default, providing up to 28 bits of randomness (8 for i386). For the stack an extra offset of 8 random bits is added. Earlier implementations made the error of only randomizing the stack, but nowadays heap and libraries are randomized as well.

Attacks on ASLR

Apply brute force. While brute-forcing the randomization offset was attractive on i386, because usually only around 8 bits of an address were random, this should no longer hold on x86-64, although yet even the 20 bits which x86-64 uses in practice could be subject to brute-forcing attacks. Shacham et al. (2004) described a method of brute-forcing a forking daemon: A `fork(2)` will not alter any randomization, thus the attacker can try to exploit with a given randomization. If he is not successful, the daemon will crash and a new child is spawned with which the attacker can try again.

Spraying. If the attacker has the option to place his data in multiple places within the process' address space, he is able to increase his likelihood of hitting the right address. This might be the case if an attacker can control environment variables or is communicating via compressed channels, that allow him to post much data without having to create unreasonable amounts of traffic. Also any scripting language allows for huge memory allocations. This is especially of interest when exploiting browser vulnerabilities, as JavaScript, Java or Flash can be used for planting data into the process' address space. In practice heap or stack spraying plants hundreds of megabytes of data and has been used in all recent browser exploits.

Partial overwrites and information leaks. If an attacker can acquire information on addresses used in the program, he or she usually has enough information to correctly guess all required addresses. Durden (2002) demonstrated that partial overwrites of the return address can slightly alter control flow although exact addresses are not known. In conjunction with format string vulnerabilities such behaviour can be used to extract stack dumps from the process providing the attacker with all information he or she might need. Such vulnerabilities are as Durden notes unlikely though.

Control the environment. During research I found out, that the randomization in a current Linux kernel only depends on the time (in a resolution of 1 / HZ seconds, which was 4ms on a testing machine) and the pid. If an attacker can locally launch the target process using `execve(2)` (which keeps the pid) just after his launching program was started, the time-frame is sufficient, so that the started process has the same randomization. Additionally the attacker can recreate the conditions for the random-number-generator with in a time-frame of up to 2 minutes (depending on the pid of target process) after the target process was started and thus get the same randomization with likelihood of probably $\frac{1}{3}$. More detailed information will be published in a seperate paper on this topic (see Fritsch, 2009).

Static pages. If for some reason not all page-mappings are randomized, this opens another door for attackers. For example Linux kernels prior to 2.6.20 (sorrow, 2008) mapped linux-gate.so to a fixed location, thus allowed the attacker to use return-into-libc techniques (cf. Section 2.2) to bypass ASLR. A brief investigation of x86-64 showed, that there as a similar flaw with the vsyscall-page being statically mapped at the address 0xffffffffff600000. Although an easy exploitation seems not to be likely because this page contains only a limited amount of useful `ret` instructions and esp. no direct `call/jmp`

`%rsp` instruction, it can be quite certain, that an attacker can find a way to subvert this page's instructions given enough time.

Additionally no code-page of an executable is randomized in the default case. This gives a much larger code-base that will be of interest for attackers. Still though, these pages are mapped to the lower address half and thus include many \0 characters, which cannot be used in most buffer overflow situations. Not randomizing the executable's pages is a major flaw that should be fixed.

2.4 Stack Smashing Protection

Since one of the key assumptions on buffer overflows used to be that they overwrite the return address in order to exploit a vulnerability, the idea of stack cookies was introduced by Cowan et al. (1998) allowing a function to check whether a stack-based buffer overflow occured. This happens with minimal performance impact, as those checks only have to be added to functions that allocate buffers on the stack. The stack cookies or canaries are guard values stored on the stack between local variables and the return address. If a buffer overflows overwriting the return address, it will also overwrite the stack cookie, which is then noticed by the current function leading to a controlled program termination. Prior to exiting, a function that is protected using a stack cookie will compare the stack cookie with the master cookie that is stored in the thread control block.

Earlier implementations had the flaw that they would only protect the return address but not the frame pointer. If an attacker can alter the frame pointer, this is also likely to be exploitable (Richarte, 2002). A current version of gcc will introduce a stack-cookie before the frame pointer and the return address, thus protecting both.

2.4.1 Bypassing stack cookies / SSP

Methods for bypassing stack cookies were first published by Kil3r and Bulba (2000) in Phrack Magazine and rely on the fact, that stack cookies do not protect local variables from being overwritten. The authors would use a buffer overflow to modify other local variables and to subvert them. In the easiest way such a local variable can be a function pointer which is called at a later time, but the more general case would be a data pointer be subsequently used, usually resulting in writing a value (ideally attacker-controlled content) to some attacker-controlled address. This is also the same mean by which heap overflows were exploited (cf. Anonymous, 2001; Kaempf, 2001). Such write32/write64 vulnerabilities can usually be exploited, for example by altering function tables like the

Global Offset Table (GOT). In case additional protection methods are in place, more creativity might be required.

Since overwriting variables was noticed to be a problem, IBM (2005) produced another gcc patch with a technique called ProPolice that will order variables on the stack in such a way, that arrays come last so that situations in which buffer overflows overwrite local pointer variables but not the canary are not possible anymore. There are however few cases in which applying this technique is not pratical, so for example for structures that define a certain variable ordering.

Unprotected buffers. For performance reasons the stack smashing protection is not applied to all functions. gcc for example will only protect functions involving char arrays (but not short, int or pointer arrays). Additionally the char buffer needs to be bigger than 4 elements, otherwise the function remains unprotected. The heuristic approach oversees quite a few vulnerable buffers and as such there were vulnerabilities like the animated cursor bug on Windows (CVE-2007-0038) or the mod_rewrite bug (CVE-2006-3747) in which buffer overflows could be easily exploited because the function was not protected.

Information leaks. If the frame pointer is not protected, then attacks as described in Section 2.3 become possible. Those can be used leak the cookie, so that an attacker can craft an exploit that will put the right cookie into the stack. The same applies if classic format string vulnerabilities allow to read values on the stack. Format string vulnerabilities (see scut, 2006 for details) make this possible if the attacker can supply crafted format strings, which are frequently used with functions like [vsf]*printf. These functions expect an arbitrary number of parameters based on the format string and misuse can lead to information leaks, stack corruption or arbitrary memory writes. However such misuse is easy to spot due to the very typical design of functions that use these format strings and thus such bugs are unlikely. This is also due to the general awareness of format string vulnerabilities and automatic detection methods.

Guessing the canary. Hawkes presented a side-channel timing attack for guessing the canary value on the stack at RuxCon 2006 and was able to reduce the search space from 2^{32} to 1024 by guessing each byte separately and using subtle timing differences to detect the correct byte. If such granular time measurements are possible and the attacker has the opportunity to try several times (e.g. due to a forking daemon), this attack is an option.

Implementation weakness During research it turned out, that the glibc-part responsible for initialising the stack-canary is actually disabled for performance-reasons as it needs to open a file descriptor and read from `/dev/random`. In this case the canary is initialised with a static value of `0xff0a000000000000` leaving doors wide open for attackers. An unofficial patch is included in several Linux-distributions though, which initialises the register based on some pseudo-random values (see python pseudo-code in Listing 1).

Listing 1: Calculation of the canary with "poor man's randomization patch"

```
def canary():
    __WORDSIZE = 64
    ret = 0xff0a000000000000
    ret ^= (rdtsc() &  0xffff) << 8
    ret ^= (%rsp   & 0x7ffff0) << (__WORDSIZE - 23)
    ret ^= (&errno & 0x7fff00) << (__WORDSIZE - 29)
    return ret
```

If ASLR guessing bugs as the one described in Fritsch (2009) are available to the attacker, then he or she knows already the stack-address and errno's address, which on a given system depends solely on the randomization bits. 16 bits of uncertainty remain from the TSC-register which are unlikely to be reduced further, but since the canary is not reinitialised upon a `fork(2)`, brute force is a likely successful option.

This flaw is going to be fixed as soon as the kernel provides an application with an initialisation vector of random data, that can then be used to set up the canaries securely without performance loss due to reading from `/dev/random`.

3 Summary and conclusions

The previous sections described modern buffer overflow mitigation techniques that are widely deployed on current not only Linux-based operating systems. Each measure has its point and renders whole classes of exploits useless, but attackers are creative and there are ways to circumvent each technique. However, if looking at the combined picture of those mitigation techniques, exploiting buffer overflows becomes *really* hard, if not impossible. If a vulnerability is discovered in an application, that has all the measures in place, than this vulnerability has to grant the attacker a way to bypass the stack cookies protection. This might be due to one of the implementation flaws, although these are going to be fixed some day. The remaining exploitable bugs are usually write32/16/64 possibilities, which thanks to NX (cf. Section 2.2) do not yet allow for direct code execution. An attacker would therefore need multiple shots or use the vulnerability as a key to opening another application-specific attack (like Dowd's famous Flash vulnerability CVE-2007-0071 that knocked off Flash's byte-code verifier opening the door for exploitation). This again might already become infeasible if ASLR (cf. Section 2.3) is in place and none of the circumvention methods described are applicable. Thus systems that are protected by these measures are, while not being immune to buffer overflows, reasonable secure against upcoming vulnerabilities.

While buffer overflows used to be simple and people could easily learn to exploit these vulnerabilities, writing reliable exploits has become an art nowadays, since the odds are just not on the side of the attacker any more. The picture has changed, the number of people knowing about the details and implications of buffer overflows and mitigation techniques is rather decreasing due to the complexity of the topic. Writing exploits requires thorough technical understanding and a huge investment of time. The outside world is even far more complex than depicted in this paper: Besides Linux there are other operating systems in different versions with similar technologies, but constrained due to other reasons. Probably less people are going to be researching specific buffer overflows, which might in turn lead to more unknown vulnerabilities, although that is sole speculation.

To draw a final conclusion on the topic: Most of the flaws of the mitigation techniques can and will be fixed and thus the importance or impact of buffer overflows will eventually decrease and will be replaced by other threats to computer security with more dangerous potential.

References

Anonymous 2001

ANONYMOUS: Once upon a free(). In: *Phrack* 11 (2001), Nr. 57. `http://www.phrack.org/archives/57/p57_0x09_Once%20upon%20a%20free()_by_anonymous%20author.txt`

Chander et al. 2007

CHANDER, A. ; ESPINOSA, D. ; ISLAM, N. ; LEE, P. ; NECULA, G. C.: Enforcing resource bounds via static verification of dynamic checks. (2007). `http://www.cs.berkeley.edu/~necula/Papers/res_esop05.pdf`

Cowan et al. 1998

COWAN, C. ; PU, C. ; MAIER, D. ; HINTONY, H. ; WALPOLE, J. ; BAKKE, P. ; BEATTIE, S. ; GRIER, A. ; WAGLE, P. ; ZHANG, Q.: StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. (1998), p. 5–5

Cowan et al. 2000

COWAN, C. ; WAGLE, P. ; PU, C. ; BEATTIE, S. ; WALPOLE, J.: Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade. (2000), p. 23–26

Dowd et al. 2006

DOWD, M. ; MCDONALD, J. ; SCHUH, J.: *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities.* Addison-Wesley Professional, 2006

Durden 2002

DURDEN, Tyler: Bypassing PaX ASLR protection. In: *Phrack Magazine* 11 (2002), Nr. 59. `http://www.phrack.com/archives/59/p59_0x09_Bypassing%20PaX%20ASLR%20protection_by_Tyler%20Durden.txt`

Fritsch 2009

FRITSCH, Hagen: Exploiting the random number generator to bypass ASLR. (2009)

Gough 2004

GOUGH, Brian: *Memory bounds-checking for GCC.* Supplement article to "An Introduction to GCC". `http://www.network-theory.co.uk/articles/boundschecking.html`. Version: May 2004

Hawkes 2006
HAWKES, Ben: Exploiting OpenBSD. (2006). `http://www.ruxcon.org.au/files/2006/hawkes_openbsd.pdf`

Kaempf 2001
KAEMPF, M.: Smashing The Heap For Fun And Profit. In: *Phrack Magazine* 11 (2001), Nr. 57. `http://doc.bughunter.net/buffer-overflow/heap-corruption.html#mem_alloc`

Kil3r and Bulba 2000
KIL3R ; BULBA: Bypassing StackGuard and StackShield. In: *Phrack* 10 (2000), Nr. 56. `http://www.phrack.com/archives/56/p56_0x05_Bypassing%20StackGuard%20and%20StackShield_by_Kil3r%20\&%20Bulba.txt`

Levy 1996
LEVY, E.: Smashing the stack for fun and profit. In: *Phrack Magazine* 7 (1996), Nr. 49. `http://www.phrack.org/archives/49/p49_0x0e_Smashing%20The%20Stack%20For%20Fun%20And%20Profit_by_Aleph1.txt`

Nergal 2001
NERGAL: Advanced return-into-lib(c) exploits (PaX case study). In: *Phrack* 11 (2001), Nr. 58. `http://www.phrack.org/archives/58/p58_0x04_Advancedreturn-into-lib(c)exploits(PaXcasestudy)_by_nergal.txt`

Richarte 2002
RICHARTE, Gerardo: Four different tricks to bypass stackshield and stackguard protection. (2002). `http://www.coresecurity.com/content/four-different-tricks-to-bypass-stackshield-and-stackguard`

scut 2006
SCUT: Exploiting Format String Vulnerabilities. (2006). `http://doc.bughunter.net/format-string/exploit-fs.html`

Shacham et al. 2004
SHACHAM, H. ; PAGE, M. ; PFAFF, B. ; GOH, E. J. ; MODADUGU, N. ; BONEH, D.: On the Effectiveness of Address-Space Randomization. (2004). `http://www.cs.jhu.edu/~rubin/courses/fall04/asrandom.pdf`

sorrow 2008
SORROW: ASLR bypassing method on 2.6.17/20 Linux Kernel. (2008). `http://packetstormsecurity.org/papers/bypass/aslr-bypass.txt`