Smashing the stack - A 25 year retrospective

Jussi Mäki
Department of Computer Science
University of Helsinki
Helsinki, Finland
joamaki@gmail.com

ABSTRACT
Buffer overflows still remain a problem for software today. Even with address space randomization and non-executable stacks software remains vulnerable to clever exploits.

In this paper we introduce the buffer overflow through a practical example and take a retrospective look on the past 25 years of the arms race on buffer overflow exploits and protections.

Categories and Subject Descriptors
D.4.6 [Operating Systems]: Security and Protection

General Terms
Security, algorithms

Keywords
Buffer overflow, Smash the stack, ASLR, return oriented programming

1. INTRODUCTION
The stack-based buffer overflow and the mangling of the return addresses in the call stack still remains a common security vulnerability even with recent defense mechanisms. This survey is a tutorial and a retrospective into the world of buffer overflow exploits and defenses against it from the past 25 years. We will start off with some definitions and a practical example for the experienced reader of a buffer overflow and its exploit. After the introduction we follow up with a brief look at some of the major exploits that have used a buffer overflow, followed up with a look at some of the successful defenses against them.

The hacker term for the act of exploiting the buffer overflow is often called “smashing the stack”. The hacker jargon [1] gives a definition:

\textit{smash the stack} [C programming] n.

On many C implementations it is possible to corrupt the execution stack by writing past the end of an array declared auto in a routine. Code that does this is said to smash the stack, and can cause return from the routine to jump to a random address. This can produce some of the most insidious data-dependent bugs known to mankind. Variants include trash the stack, scribble the stack, mangle the stack; the term mung

Let us decrypt this definition a bit.

The stack is a commonly used data structure for storing items in a last-in, first-out fashion. That is the the first item you \textit{push} in is the last to \textit{pop} out. A programming language such as C uses it for temporary storage of not only local variables, but for implementing function calls by storing the return addresses (the instruction in the caller after the function call). In function calls the called function will return back to the caller by taking the address from the stack and jumping to it.

A buffer is an array of elements such as characters or integers meant for temporary storage of raw data. For example for reading a fixed-size datum from a data file, which is then further parsed from the buffer into a more structured object. A buffer overflow is an unintentional writing past the boundaries of a buffer. Since buffers in C are often used haphazardly and the standard library functions that operate on them are often unsafe in that they do not enforce bounds checking the chances of introducing a buffer overflow is quite high, especially for a novice programmer.

![Figure 1: Process memory for typical C program](image-url)

Since languages such as C use a single stack for local variables and for implementing function calls makes it possible to divert the flow of control away to an attacker specified location. In Figure 1 above is depicted a typical C program’s process memory map. The stack is positioned at the top of
memory and grows toward lower memory address. External libraries and program code are positioned at lower memory addresses. Looking at the layout of a typical stack frame one can see that locally allocated function variables are at a lower memory address than the functions return address and overflows in any local variables may corrupt the return address.

Next we will look at an example of a vulnerable program and how it may be exploited.

**Program 1** A vulnerable C program

```c
int main() {
    char buffer[256];
    gets(buffer);
}
```

In the above program the standard library function `gets`, which reads a line from the standard input stream, does not perform bounds checking and will gladly write past the buffer’s boundaries. If the attacker writes a line longer than the buffer can hold the stack will be corrupted and the main function can be made to return to a location controlled by the attacker.

A common way of exploiting this is with a “shell code”, a small program that executes the UNIX shell. This program is copied to the target buffer and the buffer is then overflown, corrupting the stack and overwriting the function return address. By overflowing the buffer with addresses of the buffer the attacker can cause the target program to jump to the shell code placed in the beginning of the buffer and thus launch the UNIX shell (or anything else) with program’s permissions.

**Program 3** Exploit against the example program

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
/* The x86 shell-code: executes /bin/sh */
char shellcode[] =
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0
\x88\x46\x07\x89\x46\x0c\xb0\x0b
\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\n\xc0";
#define NOP 0x90
int main(int argc, char *argv[]) {
    int i;
    int bsize = atoi(argv[1]);
    char buf[bsize];
    char *p = buf;
    unsigned long addr = strtoul(argv[2], NULL, 16);
    /* Fill buffer with NOP instructions */
    memset(buf, NOP, sizeof(buf));
    /* Fill end of buffer with the buffer address */
    unsigned long *addrp = p + bsize/2;
    while (addrp < p+bsize) *addrp++ = addr;
    /* Fill in shell code */
    memcpy(p + bsize/2, shellcode, sizeof(shellcode));
    /* Write buffer to standard output */
    write(1, buf, bsize);
}
```

The above shell code can be compiled to machine code and then be injected into the target buffer. In practice though shell codes are hand assembled to be as small as possible in size and to avoid null bytes in the machine code (the null bytes signify end of string and might cause the shell code to be only partially copied to the buffer). We will use a pre-assembled shell code in the example exploit.

One last thing to piece together in order to exploit the example program is to guess the address of the buffer we want to overflow in order to overwrite the function return address. Without the protection of some of the recent defenses these addresses are mostly the same on consecutive runs of the program so it is easy to analyze the program with a debugger to find the address of the buffer.

Here the buffer we want to overflow is allocated at address `0xffffd660` and few words after the buffer is the return address of the main function (`0xf7e8ac46` at address `0xffffd75c`) which points to the initialization code of the standard C library.

Now the exploit is as simple as writing a program that constructs the right input to the target program. To increase our chances we pad the start of the buffer with NOP instructions.

```
#include <unistd.h>
#include <stdlib.h>
/* The x86 shell-code: executes /bin/sh */
char shellcode[] =
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0
\x88\x46\x07\x89\x46\x0c\xb0\x0b
\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\n\xc0";
#define NOP 0x90
int main(int argc, char *argv[]) {
    int i;
    int bsize = atoi(argv[1]);
    char buf[bsize];
    char *p = buf;
    unsigned long addr = strtoul(argv[2], NULL, 16);
    /* Fill buffer with NOP instructions */
    memset(buf, NOP, sizeof(buf));
    /* Fill end of buffer with the buffer address */
    unsigned long *addrp = p + bsize/2;
    while (addrp < p+bsize) *addrp++ = addr;
    /* Fill in shell code */
    memcpy(p + bsize/2, shellcode, sizeof(shellcode));
    /* Write buffer to standard output */
    write(1, buf, bsize);
}
```

With the above program and the size and location of the target buffer in hand we can execute our target program with input from our exploit piped to the target as shown in trace 2.

The targets small buffer now contains the shell code and...
just after the end of buffer we have rewritten the return address of the function with the buffer’s address and the target program obediently has jumped to our shell code and executed the system shell.

If a buffer overflow attack is possible remotely against for example a HTTP server the attacker can gain remote access to a machine. Even more insidious attack will target applications with setuid privileges. These applications have been given the privilege to change their currently effective user id to that of the owner of the program file. This mechanism is meant to allow non-privileged users to perform actions that require superuser access, such as mounting new filesystems or logging in as a different user. If a program with a setuid privilege is exploited the attacker can achieve superuser privileges.

Fortunately modern operating systems are no longer vulnerable to this specific attack since they come enabled with security features such as non-executable stacks and address space randomization that causes the stack to start from a random position and thus making it harder to guess the buffer addresses. We will return to these defenses and ways to break them later. ¹

2. HISTORY

Buffer overflows are not by any means a recent discovery. The technique’s publicized history goes back to 1972 where in the “Computer Security Technology Planning Study” by Electronic Systems Division noted that user could inject code to take control of the machine [2].

The first widespread exploitation of the buffer overflow vulnerability was made by the infamous Morris worm of 1988 that was estimated to have infected over 10 percent of the computers connected to the internet of the time (around 6000 machines). The worm exploited a buffer overflow vulnerability in the finger service that was enabled on most UNIX systems of the time. The author Thomas Morris became the first person to be prosecuted and convicted under a new law for computer abuse.

In 1996 Elias Levy, alias Aleph One, published the article “Smashing The Stack For Fun and Profit” which became the definitive guide to writing exploits for buffer overflows. Hundreds of exploits followed as many buffer overflows were identified and exploited in widely deployed internet services such as Sendmail, Bind and Apache. Sendmail in particular contained a large number of buffer overflow vulnerabilities [17] and was the pain of many system administrators in the late 1990s with yours truly included.

In 2001 a worm called Code Red spread through a buffer overflow vulnerability in Microsoft’s IIS server. It was reported to have infected over 359,000 hosts. It defaced the web sites served by the affected host and launched denial of service attacks. This was followed by the Blaster worm in 2003 which spread through a buffer overflow vulnerability in the DCOM RPC server in Microsoft’s Windows XP and 2000 operating systems. Also in 2003 the SQL Slammer worm targeting the Microsoft SQL server spread to over 75000 machines within ten minutes. The worm’s aggressive spreading mechanism caused many critical internet routers to crash under the high traffic load which in turn caused a flood of routing table updates in core routers that brought parts of the internet to a crawl. And all this within minutes of its launch [11].

Just counting the high-profile internet worms that have exploited a buffer overflow vulnerability the economic losses will have been in the hundreds of millions of euros. So the incentive to fix the issue surely exists. Next section looks at defenses now widely deployed in modern operating systems.

3. COUNTERMEASURES

While buffer overflows may be avoided by having well tested, correct code and avoiding non-bound checked standard library functions they still rely on the programmer to avoid making mistakes. Higher level languages such as Python or Java do bound checking on behalf of the programmer and do not suffer from buffer overflow vulnerabilities. Switching to a safer language may work for some applications, but for some the performance and portability of C and C++ remain a necessity.

To make C and C++ safer against stack buffer overflows protections were made available in the major compilers. The first defense came with StackGuard [5] which was released in 1997 and protected against overflows through a compiler extension that added a random “canary” word (chosen when C library is initialized) to the stack that was inserted at function prologue and checked at epilogue before jumping to the return address. If the attacker had no way of deducing the chosen canary the program would crash if the attacker guessed wrong. StackGuard was later enhanced and released as ProPolice SSP which was corporated into many Linux and BSD variants [9]. Most of the modern compilers today support a variant of the StackGuard approach. The GNU C compiler ships with ProPolice SSP which can be enabled through the -fstack-protector flag. Due to backward compatibility and fear of performance degradation these are not as widely used as they should be. OpenBSD which has had ProPolice SSP enabled since 2002 and claim only a 1.3% performance hit [7], which would suggest that there are not many reasons to not have it enabled by default.

To get past the stack guard protection the attacker must either resort to finding an information leak that will reveal the canary or to brute force the value. A type of timing attack can be used to brute force the canary byte at a time.

Another widely deployed way of making buffer overflows harder to exploit is to randomize the address space layout of the program so that guessing the address of the injected code becomes harder or impossible. Address-space layout randomization (ASLR) first originated from the PaX project in 2001 and from there found its way into current operating

¹In case you want to try these you’ll need to enable executable stacks with execstack tool and disable address space randomization with setarch tool.
systems.

In addition to these modern processors come with support, the “NX” bit, for disabling execution of certain areas of memory. Most operating systems now use the NX bit to mark stacks and other writable memory areas non-executable (PaX in Linux, W+X in OpenBSD, DEP in Windows). This makes code injection mostly a fruitless effort as the areas the attacker can write to are non-executable and the executable areas are read-only.

While these countermeasures make buffer overflows harder to exploit, they still leave many attack vectors undefended. Stack guards do not help against buffer overflow exploits that target function pointers or C++ vtables. Address space layout randomization provides only a probabilistic defense by making it harder to guess the location of the injected code. And finally the NX bit makes the injected code non-executable, but injecting code is not the only way of exploiting a buffer overflow to control the target. Next we will look at attacking without code injection.

4. RETURN-INTO-LIBC ATTACK

Modern defenses that make stacks non-executable and randomized makes code injections a practically impossible, but nothing prevents the attacker to make the target function return to existing executable code section. The return-into-libc attack uses the existing C library functions to control the target. This attack was discussed already in 1997 on the linux kernel mailing list and finally exploited by Solar Designer [8].

With just the NX bit to defeat an attacker can use functions in the standard library to perform arbitrary computation. For example the system() function which executes external programs takes only a single argument and which on most C compilers targeting the x86 compiler is passed in stack. So all that is needed is to overflow the buffer so that the location of the system() function, followed by a dummy return address and the argument pointing to a string containing the program we want to execute. Here we can either have the string in the buffer or use existing strings in the programs address space (the environment variables handily contain SHELL=/bin/sh) [3]. On modern 64-bit architectures this breaks as the function call parameters are passed in registers. However small snippets that place the top of stack to a register and jump to next location on stack are abundant even in small programs. We will return to this in the next section.

Address-space layout randomization makes the task of finding the locations of existing functions harder since functions are linked in to random locations. However the relative offsets of the functions remain the same and if the location of a single function from for example the C library is known then the rest can be deduced. Shacham et al. came up with a derandomization attack in their paper “On the Effectiveness of Address-Space Randomization” [16]. They show the ineffectiveness of ASLR by attacking a PaX protected Apache server with a known buffer overflow vulnerability. By repeated guessing of the address for the standard library function usleep() they can deduce a correct guess when the connection hangs instead of closing the connection. Then it is easy to call system() as its offset in relation to usleep is known. Since on 32-bit machines PaX only randomizes 16 bits of the shared library link address the attack needs at most only 65535 tries to succeed. For 64-bit machines this type of attack is much harder since the search space is orders of magnitudes larger.

5. RETURN-ORIENTED PROGRAMMING

Return-oriented programming extends the idea of calling existing code as done in return-to-libc. The idea behind it is simple: since we cannot place executable code in the overflowed buffer nor can we guess the address of a buffer we must find pieces executable code somewhere else. The technique was first introduced 2005 in paper by Sebastian Krahmer on borrowed code chunk exploitation [10]. His work was extended by Shacham in 2007 with tools to find gadgets and to automate the construction of an exploit [15]. Shacham’s work was extended by Schwartz et al. with Q, a tool to automatically craft return-oriented programming payloads.

In this technique the attacker exploits a large sequence of small snippets of code, “gadgets”, that end in the ret instruction. The ret instruction on x86 pops an address of the stack and starts executing instructions from that address. Shacham [15] showed that these tiny gadgets form a Turing-complete “instruction set” by just using the machine code from the standard C library (the GNU libc).

The above figure 5 shows a stack with a buffer overflow that has overwritten the function return address and several words beyond that. Each of these words point to a gadget that resides in either one of the linked libraries or the program code itself. When each gadget is executed the last ret instruction of each gadget takes the address of the next gadget from the stack and transfers control to it.

To make this more concrete we will look at a few example gadgets to deal with simple arithmetic and loops. For integer addition Shacham’s tool found the following sequence:

```
addl (%edx), %eax
push %edi
```
To the detriment of this and other ad-hoc methods that rely on the use of ret instruction Checkoway et al. [4] showed that ROP can be achieved without the use of a return instruction by using a suitable sequence of pop and jmp instructions as a trampoline for gadgets that would be chained not with a return instruction but with indirect jumps through the trampoline. The ad-hoc methods relying on a certain approach to ROP are thus doomed to fall behind as attackers shift strategies.

Another approach to defeating ROP is to eliminate gadgets from the resulting binaries by extending the compiler. In paper by Onarlioglu et al. [12] they present “G-Free", a pre-processor for the GNU assembler that eliminates all possible gadgets from the resulting binaries by eliminating all unaligned “free-branch” instructions and then protecting the valid aligned free-branch instructions from misuse. The free-branch instructions are the already much mentioned ret and all forms of jmp and call that allow branching to any parts of the program. The instructions are unaligned when they are not part of the correct program but can be reached by interpreting them from an unintended offset (the x86 architecture has variable length instructions that are not word-aligned). G-Free eliminates unaligned free-branches with code transformations, “alignment sleds”, by inserting NOP instructions (no operation) before an unaligned return instruction which causes the execution to realign with the intended instructions. To protect from misuse of aligned free-branch instructions G-Free incorporates a similar canary that is implemented by StackGuard. The disadvantages of this novel approach is increased code size (glibc increased by 30%) and that all libraries used by a program must be protected with G-Free.

Address-space layout randomization seems like another good method against the ROP exploits. Its effectiveness however is limited as modern operating systems only randomize parts of the address-space and many of the libraries used are not compiled as position independent executables (i.e. without absolute addressing) which makes them incompatible with ASLR. Schwartz et al. [14] in their paper “Q: Exploit hardening Made Easy” present a tool called Q to automate the construction of ROP payloads that work even in the presence of ASLR and W+X. With Q the exploit author describes the wanted payload in a high-level language called QooL that then compiles the payload out of the available gadgets in the target program. They tested their tool against programs in /usr/bin on a Ubuntu 9.10 machine and found that their tool could produce payloads that work in 80% of the binaries even with ASLR enabled. Only programs that had all their libraries compiled as position independent executables were immune to Q.

A more promising approach by Pappas et al. [13] applied in-place code randomization in order to transform Windows PE executables and libraries to break gadgets without breaking the program. This combined with ordinary ASLR gives a high probability of breaking most gadgets (they quote breaking 80% of the gadgets in their tests) and thus breaking exploits. Their method is safely applicable to third-party Windows applications and introduces no runtime overhead. To achieve gadget-breaking code transformation of arbitrary Windows applications the approach they employed only per-
formed simple code transformations that did not alter the location or the size of functions and data in the executable and thus could be applied without any symbolic debugging information and would not break applications that relied on precomputed offsets. Due to the above requirement only some of gadgets can be transformed. The practical example they gave with an existing exploit against Acrobat Reader managed to alter six out of eleven gadgets, but still effectively eliminating the exploit. Approaches such as theirs add an additional layer of security where desired, but may not widely deployable due to lack of portability and ease of use.

6. CONCLUSIONS

In this survey we introduced the buffer overflow attack, high-profile attacks exploiting the vulnerability and defenses against it. The recent papers on return-oriented programming showed that the defenses such as non-executable memory that were thought to have been effective in making buffer overflows unexploitable were in fact easily circumvented. The defenses against return-oriented programming are mostly concerned with binary modification to strip exploitable “gadgets” out from the shipped executables or by adding heavy instrumentation to the binaries to check invariants such as the LIFO structure of the stack. Even these defenses have been shown inadequate and/or unlikely to be widely deployed due to concerns over runtime overhead. This suggests that totally eradicating buffer overflows and attacks based on them from legacy languages is probably unachievable.

For most parts it is better to address the occurrence of buffer overflows by designing better tools and safer languages. Luckily the use of systems languages such as C and C++ is in decline in non-performance critical systems and most novice programmers opt to write applications in higher-level languages such as Java or Python that are mostly safe from the accidental buffer overflows we have described.

References