Overview Over Attack Vectors and Countermeasures for Buffer Overflows

Valentin Brandl
Faculity of Computer Science and Mathematics
OTH Regensburg
Regensburg, Germany
valentin.brandl@st.oth-regensburg.de
MatrNr. 3220018

***Abstract*—TODO**
***Index Terms*—Buffer Overflow, Software Security**

I. Motivation

When the first programming languages were designed, memory had to be managed manually to make the best use of slow hardware. This opened the door for many kinds of programming errors. Memory can be deallocated more than once (double-free), the programm could read or write out of bounds of a buffer (information leaks, buffer overflows (BOFs)). Languages that are affected by this are e.g. C, C++ and Fortran. These languages are still used in critical parts of the worlds infrastructure, either because they allow to implement really performant programms, because they power legacy systems or for portability reasons. Scientists and software engineers have proposed lots of solutions to this problem over the years and this paper aims to compare and give an overview about those.

Reading out of bounds can result in an information leak and is less critical than BOFs in most cases, but there are exceptions, e.g. the Heartbleed bug in OpenSSL which allowed dumping secret keys from memory. Out of bounds writes are almost always critical and result in code execution vulnerabilities or at least application crashes.

In 2018, 14% (2368 out of 16556) [1] of all software vulnerabilities that have a CVE assigned, were overflow related. This shows that, even if this type of bug is very old and well known, it's still relevant today.

II. Background

A. Technical Details

Exploitation of BOF vulnerabilities almost always works by overriding the return address in the current stack frame, so when the `RET` instruction is executed, an attacker controlled address is moved into the instruction pointer register and the code pointed to by this address is executed. Other ways include overriding addresses in the procedure linkage table (PLT) of a binary so that, if a linked function is called, an attacker controlled function is called instead, or (in C++) overriding the vtable where the pointers to an object's methods are stored.

A simple vulnerable programm might look like this:

```
int main(int argc, char **argv) {
  char buf[50];
  for (size_t i = 0; i < strlen(argv[1]); i++) {
    buf[i] = argv[1][i];
  }
  return 0;
}
```

A successful exploit would place the payload in the memory by supplying it as an argument to the programm and eventually overwrite the return address by providing an input > 50 and therefore writing out of bounds. When the `return` instruction is executed, and jumps into the payload, the attacker's code is executed. This works due to the way, how function calls on CPUs work. The stack frame of the current function lies between the two pointers base pointer (BP) and stack pointer (SP) as shown in 1. When a function is called, the value of the BP, SP and instruction pointer (IP) is pushed to the stack (Fig. 2) and the IP is set to the address of the called function. When the function returns, the old IP is restored from the stack and the execution continues from where the function was called. If an overflow overwrites the old IP (Fig. 3), the execution continues in attacker controlled code.

data1	0xFE	<- SP
data1	0xFF	<- BP

Fig. 1: Stack layout before function call

This is only one of several types and exploitation techniques but the general idea stays the same: ovewrite the return address or some kind of function pointer (e.g. in vtables or the PLT) and once that function is called, the execution flow is hijacked and the attacker can execute arbitiary code.

data2	0xF9	<- SP
data2	0xFA	<- BP
[old IP]	0xFB	
*0xFE	0xFC	
*0xFF	0xFD	
data1	0xFE	
data1	0xFF	

Fig. 2: Stack layout after function call

data2	0xF9	<- SP
[payload]	0xFA	<- BP
[controlled IP]	0xFB	
*0xFE	0xFC	
*0xFF	0xFD	
data1	0xFE	
data1	0xFF	

Fig. 3: Stack layout after overflow

B. Implications

III. Concept and Methods

A. Methods

This paper describes several techniques that have been proposed to fix the problems introduced by BOFs. The performance impact, effectiveness (e.g. did the technique actually prevent exploitation of BOFs?) and how realistic it is for developers to use the technique in real-world code (e.g. is incremental introduction into an existing codebase possible). In the end, there is a discussion about the current state.

B. Runtime Bounds Checks

The easiest and maybe single most effective method to prevent BOFs is to check, if a write or read operation is out of bounds. This requires storing the size of a buffer together with the pointer to the buffer and check for each read or write in the buffer, if it is in bounds at runtime. Still almost any language that comes with a runtime, uses runtime checking.

C. Prevent/Detect Overriding Return Address

Since most traditional BOF exploits work by overriding the return address in the current stack frame, preventing or at least detecting this, can be quite effective without much overhead at runtime. Chiueh, Tzi-cker and Hsu, Fu-Hau describe a technique that stores a redudnant copy of the return address in a secure memory area that is guarded by read-only memory, so it cannot be overwritten by overflows. When returning, the copy of the return address is compared to the one in the current stack frame and only, if it matches, the `RET` instruction is actually executed [2]. While this is effective against return oriented programming (ROP) based exploits, it does not protect against vtable overrides.

An older technique from 1998 proposes to put a canary word between the data of a stack frame and the return address [3]. When returning, the canary is checked, if it is still intact and if not, a BOF occurred. This technique is used in major operating systems but can be defeted, if there is an information leak that leaks the cannary to the attacker. The attacker is then able to construct a payload, that keeps the canary intact.

D. Restricting Language Features to a Secure Subset

E. Static Analysis

F. Type System Solutions

Condit, Jeremy and Harren, Matthew and Anderson, Zachary and Gay, David and Necula, George C. propose an extension to the C type system that extends it with dependent types. These types have an associated value, e.g. a pointer type can have the buffer size associated to it. This prevents indexing into a buffer with out-of-bounds values. This extension is a superset of C so any valid C code can be compiled using the extension and the codebase is improved incrementally. If the type extension is advanced enough, the additional information might form the base for a formal verification.

G. Address Space Layout Randomization

Address space layout randomization (ASLR) aims to prevent exploitatoin of BOFs by placing code at random locations in memory. That way, it is not trivial to set the return address to point to the payload in memory. This is effective against generic exploits but it is still posible to exploit BOF vulnerabilities in combination with information leaks or other techniques like heap spraying. Also on 32 bit systems, the address space is small enough to try a brute-force attempt until the payload in memory is hit.

H. w^x Memory

w^x (also known as non-eXecutable (NX)) makes memory either writable or executable. That way, an attacker cannot place arbitiary payloads in memory. There are still techniques to exploit this by reusing existing executable code. The ret-to-libc exploiting technique uses existing calls to the libc with attacker controlled parameters, e.g. if the programm uses the `system` command, the attacker can plant `/bin/sh` as parameter on the stack, followed by the address of `system` and get a shell on the system. ROP (a superset of ret-to-libc exploits) uses so called ROP gadgets, combinations of memory modifying instructions followed by the ret instruction to build instruction chains, that execute the desired shellcode. This is done by placing the desired return addresses in the right order on the stack and reuses the existing code to circumvent the w^x protection.

IV. Discussion

A. Ineffective or Inefficient

1) ASLR: ASLR has been really effective and wildly used in production. It is included in most major operating systems [5]. Some even use kernel ASLR [6]. Since this mechanism is active at runtime, it does not require any changes in the code itself, the programm only has to be compiled as a position-independent executable (PIE).

2) w^x: With the rise of ROP techniques, w^x protection has been shown to be ineffective. It makes vulnerabilities harder to exploit but does not prevent anything.

3) Runtime Bounds Checks: Checking for overflows at runtime is very effective but can have a huge performance impact so it is not feasible in every case. It also comes with other footguns. There might be integer overflows when calculating the bounts which might introduce other problems.

Methods that have been shown to be ineffective (e.g. can be circumvented easily) or inefficient (to much runtime overhead)...

B. State of the Art

What techniques are currently used?

C. Outlook

V. Conclusion

While there are many techniques, that protect against different types of BOFs, none of them is effctive in every situation. Maybe we've come to a point where we have to stop using memory unsafe languages where it is not inevitable. There are many modern programming languages, that aim for the same problem space as C, C++ or Fortran but without the issues comming/stemming from these languages. If it is feasible to use a garbage collector, Go might work just fine. If real-time properties are required, Rust could be the way to go, without any language runtime and with deterministic memory management. For any other problem, almost any other memory safe language is better than using unsafe C.

VI. Sources (Dummy Section for Deadline)

- RAD: A Compile-Time Solution to Buffer Overflow Attacks [2] (might not protect against e.g. vtable overrides, PLT address changes, ...)
- Dependent types for low-level programming [4]
- StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attachs [3] (ineffective in combination with information leaks)
- Type-Assisted Dynamic Buffer Overflow Detection [7]
- On the Effectiveness of NX, SSP, RenewSSP, and ASLR against Stack Buffer Overflows [8]
- What Do We Know About Buffer Overflow Detection?: A Survey on Techniques to Detect A Persistent Vulnerability [9]
- Survey of Attacks and Defenses on Stack-based Buffer Overflow Vulnerability [10]
- Beyond stack smashing: recent advances in exploiting buffer overruns [11]
- Runtime countermeasures for code injection attacks against C and C++ programs [12]

References

[1] MITRE. (2018). Security Vulnerabilities Published In 2018(Overflow), [Online]. Available: https://www.cvedetails.com/vulnerability-list/year-2018/opov-1/overflow.html (visited on 11/10/2019).

[2] Chiueh, Tzi-cker and Hsu, Fu-Hau, "RAD: A Compile-Time Solution to Buffer Overflow Attacks," in *21st International Conference on Distributed Computing Systems*, 2001.

[3] Cowan, Crispan and Po, Calton and Maier, Dave and Walpole, Jonathan and Bakke, Peat and Beattie, Steve and Grier, Aaron and Wagle, Perru and Yhang, Qian, "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," in *7th USENIX Security Symposium*, 1998.

[4] Condit, Jeremy and Harren, Matthew and Anderson, Zachary and Gay, David and Necula, George C., "Dependent Types for Low-Level Programming," in *Programming Languages and Systems*, 2007.

[5] Konstantin Belousov. (2019). Implement Address Space Layout Randomization (ASLR), [Online]. Available: https://svnweb.freebsd.org/base?view=revision%5C&revision=r343964 (visited on 12/10/2019).

[6] Jake Edge. (2013). Kernel address space layout randomization, [Online]. Available: https://lwn.net/Articles/569635/ (visited on 12/10/2019).

[7] Lhee, Kyung-suk and Chapin, Steve J., "Type-Assisted Dynamic Buffer Overflow Detection," in *11th USENIX Security Symposium*, 2002.

[8] H. M. Gisbert and I. Ripoll, "On the Effectiveness of NX, SSP, RenewSSP, and ASLR against Stack Buffer Overflows," in *IEEE 13th International Symposium on Network Computing and Applications (ISNCA)*, 2014.

[9] Chaim, Marcos and Santos, Daniel and Cruzes, Daniela, "What Do We Know About Buffer Overflow Detection?: A Survey on Techniques to Detect A Persistent Vulnerability," in *International Journal of Systems and Software Security and Protection (IJSSSP)*, 2018.

[10] Wang, Wei, "Survey of Attacks and Defenses on Stack-based Buffer Overflow Vulnerability," in *7th International Conference on Education, Management, Information and Computer Science (ICEMC 2017)*, 2017.

[11] J. Pincus and B. Baker, "Beyond stack smashing: recent advances in exploiting buffer overruns," *IEEE Security & Privacy*, vol. 2, no. 4, 2004.

[12] Younan, Yves and Joosen, Wouter and Piessens, Frank, "Runtime countermeasures for code injection attacks against C and C++ programs," *ACM Computing Surveys (CSUR)*, vol. 44, no. 3, 2012.