Overview Over Attack Vectors and Countermeasures for Buffer Overflows

Valentin Brandl
Faculty 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). 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 buffer overflows 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.

II. Background

A. Technical Details

Exploitation of buffer overflow 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.

B. Implications

III. Concept and Methods

A. Methods

This paper will describe several techniques that have been proposed to fix the problems introduced by buffer overflows. The performance impact, effectiveness (e.g. did the technique actually prevent exploitation of buffer overflows?) and how realistic it is for the technique to be used in real-world code (e.g. can it be introduced into an existing codebase incrementally?). In the end, the current state will be discussed.

B. Runtime Bounds Checks

The easiest and maybe single most effective method to prevent buffer overflows 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.

C. Prevent/Detect Overriding Return Address

Since most traditional buffer overflow 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 and Hsu 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[1]. 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[2]. When returning, the canary is checked, if it is still intact and if not, a buffer overflow 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, Harren, Anderson, *et al.* 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 can be improved incrementally. If the type extension is advanced enough, the additional information can even be used as the base of a formal verification.

G. Address Space Layout Randomization

Address space layout randomization (ASLR) aims to prevent exploitatoin of buffer overflows 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 can still be exploited 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 is included in all major operating systems. Some even use kernel ASLR. 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 buffer overflows, 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

- RAD: A Compile-Time Solution to Buffer Overflow Attacks[1] (might not protect against e.g. vtable overrides, PLT address changes, ...)
- Dependent types for low-level programming[3]
- StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attachs[2] (ineffective in combination with information leaks)
- Type-Assisted Dynamic Buffer Overflow Detection[4]
- On the Effectiveness of NX, SSP, RenewSSP, and ASLR against Stack Buffer Overflows[5]

References

[1] T.-c. Chiueh and F.-H. Hsu, "RAD: A Compile-Time Solution to Buffer Overflow Attacks," in *21st International Conference on Distributed Computing Systems*, 2001.

[2] C. Cowan, C. Po, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Yhang, "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," in *7th USENIX Security Symposium*, 1998.

[3] J. Condit, M. Harren, Z. Anderson, D. Gay, and G. C. Necula, "Dependent types for low-level programming," in *Programming Languages and Systems*, 2007.

[4] K.-s. Lhee and S. J. Chapin, "Type-Assisted Dynamic Buffer Overflow Detection," in *11th USENIX Security Symposium*, 2002.

[5] 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*, 2014.

Acronyms

ASLR	Address Space Layout Randomization
NX	Non-eXecutable
PIE	Position-independent Executable
PLT	Procedure Linkage Table
ROP	Return Oriented Programming