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. Main Part, TODO

A. Background

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 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. Concept and Methods

1) 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.

2) Prevent 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 et al 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 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.

3) Restricting Language Features to a Secure Subset:

4) Static Analysis:

5) Type System Solutions: Condit 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.

6) ASLR: 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.

7) w^x Memory: This mitigation 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. Return oriented programming (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.

C. Discussion

1) Ineffective or Inefficient:

2) State of the Art:
text

III. Conclusion and Outlook

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.

IV. 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]

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*, R. De Nicola, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 520–535.

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