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*—This paper tries to explain the details behind buffer overflows, explore the problems stemming from those kinds of software vulnerabilities and discus possible countermeasures with focus on their effectiveness, performance impact and ease of use.**

***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), invalid pointers can be dereferenced (`NULL` pointer dereference; this is still a problem in many modern languages) or the program 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. While most if not all of these problems are solved in modern programming languages, these languages are still used in critical parts of the worlds infrastructure, either because they allow to implement really performant programs, offer deterministic runtime behaviour (e.g. no pauses due to garbage collection), 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 [1] 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) [2] 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

Code execution via BOF vulnerabilities almost always works by overwriting the return address in the current stack frame (known as "stack smashing") [3], so when the `RET` instruction is executed, an attacker controlled address is moved into the instruction pointer (IP) register and the code pointed to by this address is executed [4]. Other ways include overwriting addresses in the procedure linkage table (PLT) (the PLT contains addresses of dynamically linked library functions) of a binary so that, if a linked function is called, an attacker controlled function is called instead, or (in C++) overwriting the vtable where the pointers to an object's methods are stored.

A simple vulnerable C program might look like this:

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

Fig. 1: Vulnerable C program

A successful stack BOF exploit would place the payload in the memory by supplying it as an argument to the program (or by placing it in an environment variable, writing it to a file that the program reads, via network packet, ...) and eventually overwrite the return address by providing an input with > 50 bytes and therefore writing out of bounds. When executing the `return` instruction, and the 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 base pointer (BP) and stack pointer (SP) as shown in fig. 2a. When a function is called, the value of the BP and IP is pushed to the stack (fig. 2b) 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. 2c), the attacker controls where execution continues.

This is only one of several types and exploitation techniques. Others include

argc	0xFE	← SP (main)
argv	0xFF	← BP (main)

(a) Stack layout before function call

buf	0xC8	← SP (vuln)
buf	...	
buf	0xFA	← BP (vuln)
[old IP]	0xFB	
[BP (main)]	0xFC	
[*input]	0xFD	
argc	0xFE	
argv	0xFF	

(b) Stack layout after function call

[payload]	0xC8	← SP (vuln)
[payload]	...	
[payload]	0xFA	← BP (vuln)
[controlled IP]	0xFB	
[BP (main)]	0xFC	
[*input]	0xFD	
argc	0xFE	
argv	0xFF	

(c) Stack layout after overflow

Fig. 2: Stack layouts during an BOF exploit

- Heap-based BOF: In this case there is no way of overwriting the return address but objects on the heap might contain function pointers (e.g. for dynamic dispatch) which can be overwritten to execute the attackers code, when executed [4].
- Integer overflow: Some calculation on fixed sized integers is used to allocate memory. The calculation leads to an integer overflow and only a small buffer is allocated [4]. Later a big integer into the buffer is used and reads or writes outside the buffer. This kind of vulnerability can also lead to other problems because at least in C, signed integer overflow is undefined behaviour.

This paper won't explore other kinds of BOF in detail because the concept is always the same: Unchecked indexing into memory allows the attacker to overwrite some kind of return or call address, which allows hijacking of the execution flow.

The most trivial kinds of payloads is known as a `NOP` sled. Here the attacker appends as many `NOP` instructions before any shell-code (e.g. to invoke `/bin/sh`) and points the overwritten IP or function pointer somewhere inside the `NOP`s. The execution "slides" (hence the name) through the `NOP`s until it reaches the shell-code. Most of the mitigation techniques described in this paper protect against this kind of exploit but there are different and more complex ways of exploiting BOFs that are not that easily mitigated.

III. Concept and Methods

A. Research Methods

This paper describes several techniques that have been proposed to mitigate the problems introduced by BOFs and tries to answer the following questions:

- What is the performance impact?
- How effective is the technique? Did it actually prevent exploitation of BOFs?
- How realistic is it for developers to use the technique in real-world code? Can it be introduced incrementally?

The paper focuses on solutions for the C language, since it is still the second most used language as of December 2019 [5]. Some of the described techniques are language agnostic but this is not a focus of this paper. In the end, there is a discussion about the current state.

For the literature research, the paper "What Do We Know About Buffer Overflow Detection?: A Survey on Techniques to Detect A Persistent Vulnerability" served as a base. From there a snowball system search with combinations of the keywords "buffer", "overflow", "detection", "prevention" and "dependent typing" was performed using https://scholar.google.com/.

Results are evaluated and prioritized using the following criteria:

- Type of publication in the following order:
  1) conference paper
  2) unreleased paper
  3) books
  4) online sources
- Number of citations
- Publisher
- Author's reputation and institute
- Overall quality (first by checking structure and abstract, then by the actual content)

B. Runtime bounds checking (RBC)

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 (so called fat pointers) 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. For this technique to be effective effective in general, writes to a raw pointer must be disallowed. Otherwise the security checks can be circumvented. RBC introduces a runtime overhead for every indexed read or write operation. This is a problem if a program runs on limited hardware or might impact real-time properties.

Introducing RBC into an existing codebase is not easy. Using fat pointers in a few functions does not prevent other parts of the code to use raw pointers into the same buffer. So for this to be effective, the whole codebase needs to be changed to disallow raw pointers, which, depending on the size, might not be feasible. Still, if done correctly and consequently, it is simply impossible to exploit BOFs for code execution. Denial of service (DOS) is still possible because the program terminates gracefully when a out of bounds index is used.

C. Prevent/Detect Overwriting Return Address

Since most traditional BOF exploits work by overwriting 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 redundant 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 [6]. While this is effective against stack based BOFs, in the described form, it does not protect against vtable overwrites. An extension could be made to also protect the PLT and vtables but custom constructs using function pointers would still be vulnerable. Since this technique is a compiler extension, no modification of the codebase is required to enable it, and while it does not prevent all kinds of BOF, mitigates all stack based BOFs with only minimal overhead when calling and returning from a function.

An older technique from 1998 proposes to put a canary word (named after the canaries that were used in mines to detect low oxygen levels) between the data of a stack frame and the return address [7][8]. When returning, the canary is checked, if it is still intact and if not, a BOF occurred. This technique is implemented by major compilers [9] but can be defeated, if there is an information leak that leaks the canary to the attacker. The attacker is then able to construct a payload, that keeps the canary intact. This mitigation has a minimal performance impact [9] and offers a good level of protection. It is a compiler extension so no modification of the code base is needed.

D. 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 [10]. 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. In some cases, the type extensions can even be inferred [10].

This technique prevents all kinds of overflows, if used, but requires changes to the codebase and is only effective where these changes are applied. Since it is a compile-time solution, it does affect the compile-time but has no negative effect on the runtime.

E. Address Space Layout Randomization

Address space layout randomization (ASLR) aims to prevent exploitation of BOFs by placing code at random locations in memory [8]. That way, it is not trivial to set the return address to point to the payload in memory. This is effective against every kind of BOF vulnerability but it is still possible 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 [11].

This is another technique that works without modification of the code base. Also there is no runtime overhead because nothing changed except the location of the program.

F. w^x Memory

w^x (also known as non-eXecutable (NX) or data execution prevention (DEP)) makes memory either writable or executable [8]. That way, an attacker cannot place arbitrary 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 program 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 (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. These combinations of memory modification followed by `RET` instructions are called ROP chains and are Turing complete [12], so in theory it is possible to implement any imaginable payload, as long as the exploited program contains enough gadgets and the overflowing buffer has enough space.

IV. Discussion

A. Effectiveness

1) ASLR: ASLR has been proven effective and is wildly used in production. It is included in most major operating systems [13]. Some even use kernel ASLR [14]. Since this mechanism is active at runtime, it does not require any changes in the code itself, the program only has to be compiled as a position-independent executable (PIE). On 32-bit CPUs, only 16-bit of the address are randomized. These 16-bit can be brute forced in a few minutes or seconds [15].

There is no runtime overhead since the only change is the position of the program in memory. Since there is no additional work except maybe recompilation, this technique can and should be used on modern systems.

2) w^x: With the rise of ROP techniques, w^x protection has been shown to be ineffective. It makes vulnerabilities harder to exploit by preventing the most naive types of payloads but it doesn't actually prevent exploits from happening.

NX does not prevent any exploits but makes it harder for an attacker that does not know the system, the program is running on (e.g. a network service). It has no runtime overhead and is a compile-time option so it does not hurt to enable NX.

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 bounds which might introduce other problems.

B. State of the Art

Operating systems started to compile C code to PIE by default [16] and ASLR is enabled, too. Same goes for NX and stack canaries [16]. The combination of these mitigations makes it hard to write general exploits for modern operating systems.

To check the current state, the author investigates, which mitigations are enabled by default in the latest release (9.2) of the GNU compiler collection (GCC) and the latest commit of the LLVM-project (`181ab91efc9`) by compiling both compilers using the default configuration. The experiments are performed on a 64-bit Debian 9.11 system running on version 4.19.0 of the Linux kernel. The following commands compile the source codes:

```
mkdir objdir \
&& cd objdir \
&& ../configure \
  --build=x86_64-linux-gnu \
  --host=x86_64-linux-gnu \
  --target=x86_64-linux-gnu \
  --disable-multilib \
&& make -j8
```

(a) GCC compilation script

```
mkdir build \
&& cd build \
&& cmake -DLLVM_ENABLE_PROJECTS=clang \
  -DCMAKE_BUILD_TYPE=Release \
  -G "Unix Makefiles" ../llvm \
&& make -j8
```

(b) clang compilation script

The `build`, `host` and `target` parameters in fig. 3a describe the target platform for the compiler and `disable-multilib` disables 32-bit support. The `-j8` flag only tells make to use all 8 available cores for compilation. `CMAKE_BUILD_TYPE=Release` creates a release build of the clang compiler (see fig. 3b).

The fresh builds of GCC and clang compile the code from fig. 1 to check which mitigations are enabled by default. Using `gcc -o vuln.gcc vuln.c` and `clang -o vuln.clang vuln.c` to compile the source code, the `checksec.sh` tool [17] shows which mitigations are active in the new binary:

Mitigation	Active in GCC?	Active in clang?
Stack Canary	No	No
NX	Yes	Yes
PIE	No	No

TABLE I: Enabled mitigations in a default GCC and clang build

Surprisingly enough, two of the most popular C compilers enable only one of the described compile-time mitigations by default (see table I). Maintainer of operating system packages of the compiler might choose a more secure configuration for the compiler as shown in [16] but still, compiler vendors might want to choose better defaults, too.

So far, all discussed mitigations don't change anything about the existence of BOFs but just try to prevent the exploitation for code execution. The vulnerable programs terminate if the stack canary is overwritten, a call into NX memory occurs or execution continues inside garbage data due to ASLR. The underlying problem persists, only the worst results are mitigated. DOS is still a problem in safety critical systems (e.g. cars, planes, medical devices) or in any area with real-time requirements.

Language extensions to fix the problem of BOFs as described in [10] require lots of discipline to use them everywhere. They are only useful if the whole codebase uses the new features. Introducing them in an existing codebase is quite unrealistic since it requires lots of modifications. On the other hand, this actually prevents BOFs from happening and not just from being exploited, so it looks like an interesting concept for safety critical software.

V. Conclusion

While there are many techniques, that protect against different types of BOFs, none of them is effective in every situation but in combination they offer good protection against code execution attacks. Maybe the time has come, where usage of memory unsafe languages has to be stopped 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 coming from these languages. If it is feasible to use a garbage collector, languages like Go, Java or even scripting languages like Python 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.

References

[1] Codenomicon. (2014). The Heartbleed Bug, [Online]. Available: http://heartbleed.com/ (visited on 12/15/2019).

[2] 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).

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

[4] 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.

[5] TIOBE. (2019). TIOBE Index for December 2019, [Online]. Available: https://www.tiobe.com/tiobe-index/ (visited on 12/15/2019).

[6] 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.

[7] 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.

[8] 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.

[9] P. Wagle and C. Cowan, "StackGuard: Simple Stack Smash Protection for GCC," Immunix, Inc., Tech. Rep., 2003. [Online]. Available: ftp://gcc.gnu.org/pub/gcc/summit/2003/Stackguard.pdf (visited on 12/15/2019).

[10] 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.

[11] Gisbert, H. M. and Ripoll, I., "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.

[12] H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)," in *Proceedings of the 14th ACM conference on Computer and communications security (CCS)*, 2007.

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

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

[15] Shacham, Hovav and Page, Matthew and Pfaff, Ben and Goh, Eu-Jin and Modadugu, Nagendra and Boneh, Dan, "On the Effectiveness of Address-Space Randomization," in *11th ACM conference on Computer and communications security (CCS)*, 2004.

[16] Bartłomiej Piotrowski. (2017). 7.1.1-4: enable SSP and PIE by default, [Online]. Available: https://git.archlinux.org/svntogit/packages.git/commit/trunk?h=packages/gcc&id=5936710c764016ce306f9cb975056e5b7605a65b (visited on 12/15/2019).

[17] T. Klein. (2019). Checksec.sh, [Online]. Available: https://github.com/slimm609/checksec.sh (visited on 12/16/2019).