Introduction to Binary Exploitation

Arvind S Raj
(arvindsraj@am.amrita.edu)

CSE467 IntroSSOC

B.Tech CSE Jan-May 2017

Looking back and ahead

Till now

- Assembly programming - with C library and using system calls.
- Software reverse engineering: from assembly to higher level code.

Up next

- Exploiting (some)poorly written programs to execute arbitrary code.
- Bypassing some widely deployed defences to make exploiting harder(but not impossible).

Overview

- Objective: Given a program, determine if there's any obvious vulnerability.
- Carefully analyze binary and influence behaviour to execute arbitrary code.
- Bug finding - active research area in industry and academia.
- Working exploits for popular programs pay a lot(and hard to find)!

Outline

Understanding stack layout

Caller frame		Local variable n - 2
		Local variable n - 1
		Local variable n
		Return address
Callee frame	RBP	Saved RBP
		Local variable 1
	RSP	Local variable 2

Understanding stack layout(cont.)

- call instruction pushes address of next instruction onto stack before starting new function.
- Overwrite return address $\Longrightarrow$ gain control of execution flow.
- Carefully crafted input can enable executing any arbitrary code.

Buffer overflow

- Oldest known vulnerability - discovered over 20 years ago and still exists today.
- "Overflow" - write more contents than can be held in a "buffer"(eg: array).
- If buffer is overflown carefully, can execute arbitrary code!

Stack layout: Normal vs overflown

	Local variables of caller
	Return address
RBP	Saved RBP
	GARBAGE!
	GARBAGE!
	GARBAGE!
	IJKLMNOP
RSP	ABCDEFGH

	Local variables of caller
	AAAAAAAA
RBP	AAAAAAAA
	AAAAAAAA
	AAAAAAAA
	AAAAAAAA
	AAAAAAAA
RSP	AAAAAAAA

Buffer overflow demo

Let's actually do this in practice and exploit a vulnerable binary.

Outline

What is W⊕X?

- Flaw in previous case: memory locations can be modified and executed.
- Solution: No memory location should be both writeable and executable - Write $\oplus$ eXecute.
- Try the previous exploit on the second binary! Also, see output of *vmmap* in gdb during execution.
- But this protection can be bypassed. Can you guess how?

Code reuse attacks

- Use code in executable regions during process execution to mount an attack.
- Eg: Invoke commands using system function. Other ways to execute commands also possible.
- Problem: How do we set register values to pass arguments to system function call?

Code reuse gadgets

- **Gadgets**: Small sequences of assembly instructions that perform some specific action and terminate with a control flow instruction(call, jmp or ret).
- Example gadget: *x/3i 0x4005f1* in gdb.
- Above gadget enables controlling value of *rdi* and *rsi* i.e. first two arguments to any function.

Finding address of required functions

- Inspect process during runtime in gdb.
- *print* <*function_name*> will display address.
- Eg: *print system* in gdb. Displays address of *system* function.
- Also, use *vmmap* to determine which object does *system* function belong to.

Defeating W$\oplus$X demo

Let's actually bypass W$\oplus$X on a vulnerable program and achieve code execution.

Stack layout for NX bypass

Buffer		
Return address	G1 address	mov rbp, rsp; pop rdi; pop rsi; ret
	/bin/ls	
	64 bit value	
	G2 address	push rbp; mov rbp, rsp; pop rdi; pop rsi; ret
	64 bit value	
	system	

Outline

1. Basic buffer overflow attack
2. Defeating W⊕X
3. Defeating ASLR
4. Defending against memory corruption issues

Introduction

- A flaw in previous case: all library functions were loaded in same location $\implies$ easy to discover.
- Solution: Randomize location where library functions are loaded into memory making it harder to locate them.
- Popularly called Address Space Layout Randomization or ASLR.

What is ASLR?

- */proc/self/maps* contains the memory map of whichever process reads the file.
- Run *setarch x86_64 -R cat /proc/self/maps* thrice.
- Run *cat /proc/self/maps* thrice.
- Compare outputs in both cases.

What is ASLR?

- ASLR randomizes the location of all libraries, stack, heap etc.
- Hardcoding address no longer work because of this.
- Let's try the same exploit we used to defeat $W \oplus X$.
- But, this can also be bypassed because randomization isn't very good.

Understanding ASLR weakness

- Two key points about ASLR.
  - The *.text* section isn't randomized at all.
  - All executable components are randomized as a single unit - there is no randomization within them.
- So if we find location of 1 entry in say the library, we know where everything is in the library.
- Let's dump a few cores and verify this!

Understanding ASLR weakness

- *ulimit -c unlimited* - required to enable core dumping.
- Run *vulnerable.out* in *aslr* folder with exploit used in *nx*.
- Rename the core file(named *core*) dumped after segmentation fault. Generate 3 such core files.
- Load each core file and view address of system and printf as well as offset between system and printf.

Understanding ASLR weakness

- From core dumps, we saw address of printf and system vary but offset is a constant 56752 bytes.
- i.e. relative location of library instructions remains the same.
- Note: Offset will vary across different versions of same library.
- Problem: We need at least one address to find location of others.

Understanding run-time linking

- C library function definitions stored in shared libraries.
- Address of these functions unknown till runtime but call instruction needs valid target in binary.
- Solution: call target is a value at a fixed location. Value changes during runtime to point to correct function.
- Also enables sharing of libraries $\implies$ memory savings.
- Let's see this in action.

Global Offset Table

- Procedure Linkage Table(PLT): Library function calls go to this table.
- Global Offset Table(GOT): Set of pointers with addresses of library functions.
- Location of GOT fixed i.e. address library functions stored at fixed address!
- How can this be used to bypass ASLR?
- More info at https://www.technovelty.org/linux/plt-and-got-the-key-to-code-sharing-and-dynamic-libraries.html.

Bypassing ASLR using GOT overwrites

ASLR bypass strategy

- Address of printf known and offset from printf to system known.
- During runtime, modify printf GOT entry to point to system.
- Now print's PLT points to system's address.
- Setup arguments appropriately for system and call it via printf's PLT.

Bypassing ASLR demo

Let's actually bypass ASLR on a vulnerable program and achieve code execution.

Stack layout for ASLR bypass

Buffer		
Return address	Gadget 1	pop rdi; pop rsi; ret
	printf's GOT entry	Address of entry
	printf-system	Offset to system
	Gadget 2	sub [rdi], rsi; ret
	Gadget 3	mov rbp, rsp; pop rdi; pop rsi; ret
	/bin/ls	
	64 bit value	
	Gadget 4	push rbp; mov rbp, rsp; pop rdi; pop rsi; ret
	64 bit value	
	printf's PLT entry	system

Outline

Defending against memory corruption

- Used a programming bug and taking advantage of being able to execute whatever we want.
- Two possible approaches to dealing with issues: prevent and contain.
- **Prevent**: Ensure code doesn't have security bugs. Hard to achieve but most effective.
- **Contain**: Even if bug exists, restrict what can be done to limit damage. Easier to achieve but not quite effective.

Outline

1. Basic buffer overflow attack
2. Defeating W$\oplus$X
3. Defeating ASLR
4. Defending against memory corruption issues
   - Preventing security bugs
   - Containing exploits

Preventing security bugs

- Thorough security audits: ensure code written is not bug free.
- Requires good knowledge of security bugs and attacker mindset.
- Use tools such as fuzzers, code analyzers etc to find bugs.
- We will look at ASAN from Google.

Google's Address Sanitizer

- ASAN = Address SANitizer. Developed by Google and integrated into Clang and GCC. Works on many *nix OS including Android.
- Detects several memory errors including many kinds of buffer overflows, use after family, memory leaks and more.
- Adds significant execution overhead(2x) $\implies$ use during development/testing only.
- Alternative exist: Valgrind's MemCheck, DynamoRIO's Dr. Memory and GCC Mudflap.

Address Sanitizer in action

- *gcc -fsanitize=address vulnerable.c -o asan-gcc.out*
- *clang-4.0 -fsanitize=address vulnerable.c -o asan-clang.out*
- DEMO ONLY!! Add *-fno-stack-protector* to see ASAN in action.
- Run both these binaries with the exploit you wrote for ASLR bypass. Does it work?

More about Address and other sanitizers

- Many more sanitizers exist: Thread, Memory, Undefined behaviour and more.
- Not all seem to work on gcc; use clang instead.
- Not fool proof but can find many bugs. Chrome and Firefox use these regularly.
- Remember: use during development only.
- More info at https://github.com/google/sanitizers and https://clang.llvm.org/docs/index.html.

Fuzzers

- Provide random input to binaries to induce a crash.
- Can be quite smart and quickly produce crashes.
- Popular, free fuzzers: AFL-fuzz, libFuzzer(LLVM based) and radamsa.
- Won't be demoing this but all are freely available.

Outline

Some reasons ASLR bypass worked

- Control information(return address) stored on stack with data.
- Returning to any address after executing a function allowed.
- ASLR randomization very poor: leak 1 address and every address known.
- Everybody runs same code i.e. exploit 1 $\implies$ exploit all.

Proposed defences for previous flaws

- Data hiding: Split stack into safe and unsafe stack. Stack canaries: abort if stack overflow occurs.
- Control Flow Integrity: Restricting set of addresses return allowed to.
- Fine grained ASLR: Randomizing at finer granularity(page level, basic block level)
- Diversity: Everyone runs different, functionally equivalent code.
- Not exhaustive - more ways possible

Stack canaries

- Derives name from canaries in coal mines. Also, called stack cookies.
- Insert a value between local variables and return address. Overwritten $\implies$ stack overflow. Abort!
- Not comprehensive defence: can be bypassed if stack cookie value is known/discoverable.
- Stronger versions: *-fstack-protector-strong*(clang only) and *-fstack-protector-all*.

Data hiding

- Objective: Store control information in a hidden location away from data.
- Location of control information not guessable.
- Popular implementation: clang SafeStack. Separates stack into: safe and unsafe parts.
- Compile with SafeStack: *clang-4.0 -fsanitize=safe-stack vulnerable.c -o safe-stack.out*.
- DEMO ONLY!! Add -fno-stack-protector to see SafeStack in action.

Security of clang SafeStack

- Increases the bar for exploits.
- Not fully secure: thread spraying and allocation oracles can locate the "SafeStack".
- Above issues can be fixed $\implies$ not completely unusable.
- Read "Poking holes in information hiding" USENIX paper or watch "Bypassing clang's SafeStack for fun and profit" for SafeStack bypass.

Control Flow Integrity

- Restrict allowed addresses to return to.
- Eg: main should always return to it's caller(libc_start_main) and not much else.
- Buffer overflows typically jump to gadgets or other functions.
- Restrict this in code $\implies$ no longer possible.

Control Flow Integrity

Figure : Original CFI

```
bool lt(int x, int y) {
    return x < y;
}

bool gt(int x, int y) {
    return x > y;
}

sort2(int a[], int b[], int len)
{
    sort( a, len, lt );
    sort( b, len, gt );
}
```

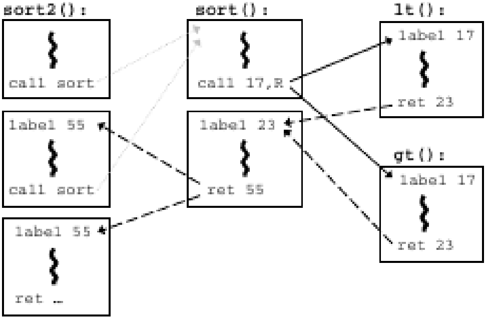

Image courtesy "Control flow integrity" paper by Abadi et al.

Control Flow Integrity issues

- Checks at every return from a function.
- Introduces high performance overhead.
- Solution: Reduce number of checks by increasing range of acceptable transfers.
- Reduced strictness on CFI $\implies$ possible to bypass using gadgets at valid return sites.

Control Flow Integrity

Figure : Loose CFI

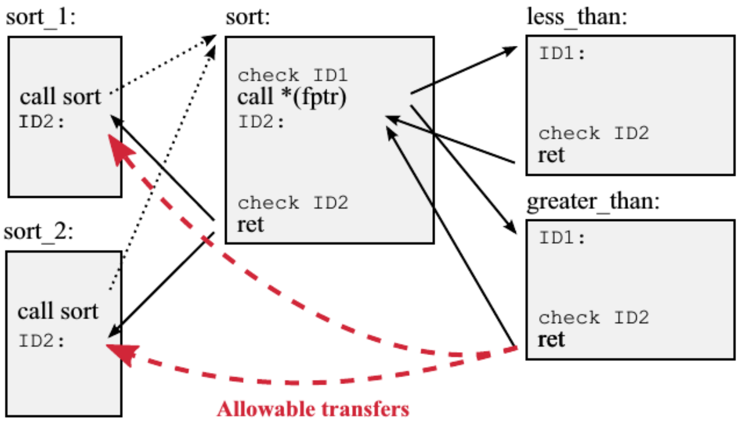

Image courtesy "Out of control: overcoming CFI" paper by Goktas et al.

Fine grained randomization

- Randomize code at finer granularity: page level, basic block level etc.
- No longer possible to compute address of function using offset to known function.
- Not secure - leak 1 page and use code pointers in it to find other code pages.
- See BlackHat 2013 talk/IEEE S&P paper on Just-in-time Code Reuse attacks for more info.

Software diversity

- Inspired by genetic diversity: everyone run different binaries. Eg: App stores.
- Gadgets no longer university $\implies$ single exploit won't work.
- Not yet proven broken but we're working on bypassing it with just 1 exploit.
- Read "SoK: Automated Software Diversity" from IEEE S&P 2014 for comprehensive overview.

Securing your C/C++ programs

- Use *-fstack-protector-all*, *-D_FORTIFY_SOURCE=2* and GOT entry protection when compiling.
- Enable available protections: clang's stack protector, CFI, sanitizers and more, PaX's grsecurity patch etc.
- Use fuzzers, static analyzers, sanitizers and thorough code review to find bugs.
- Learn about types of vulnerabilities and exploitation techniques and don't write insecure code!
- Use standard libraries and use them safely.

Conclusion

- Buffer overflows: 20 years old but still around.
- Can potentially allow hijacking control and thus, arbitrary code execution. Somewhat difficult but not impossible.
- Many more techniques exist: exploiting format string, heap overflow, exploiting heap and more.
- Some defences exist but not comprehensive and 100% effective.
- Use safer languages like Java, Python, OCaml, Haskell etc.
- If need to use C/C++, writing bug free code is the best way to prevent memory corruption issues.
- Vulnerability scanners and containment tools can

Some useful commands

- **Mark stack as executable**: *execstack -s /path/to/executable*.
- **Run a program with ASLR disabled**: *setarch x86_64 -R /path/to/executable*. gdb disables ASLR when debugging.
- **Disabling ASLR system wide**: *sysctl kernel.randomize_va_space=0*. Requires root permission.

Some useful resources

- https://clang.llvm.org/docs/ControlFlowIntegrity.html and https://blog.trailofbits.com/2016/10/17/lets-talk-about-cfi-clang-edition/.
- http://blog.quarkslab.com/clang-hardening-cheat-sheet.html.
- https://clang.llvm.org/docs/AddressSanitizer.html and other sanitizers. Also, see https://github.com/google/sanitizers/wiki.
- All research papers mentioned in the defences section of slides.