Introduction to Binary Exploitation

Lower level knowledge

- Why do hackers care?
- [source of flashing light]
- [demo of advanced technology]
- Can you turn off the light with complete understanding of the software?
- What about with basic understanding of electronics?

Even lower level knowledge

- There is no software patch against unplugging it
- Electronics “patches”:
  - Wireless power
  - Integrated battery
- Are there any attacks against a well-designed electronics system?

Today’s target

- We usually focus on getting control of the CPU
- But the CPU only does what the memory tells it to do
- Control memory → control CPU → do big hack

Paged memory

- Linear model
- OS sets up virtual address space for process
  - Maps physical RAM addresses to virtual ones
  - Hardware transparently handles this; no special userspace code required
- Pages are blocks of memory
  - Map to some block of physical memory
  - Read, Write, eXecute flags

Source demo

- [nano demo.c]
  - Runs the [date] command, which prints the date
  - Copies text to new_text
  - Prints new_text
  - Reads a line into buf
  - Prints another predermined string
  - Prints text from buf
  - Also has the dead function that isn’t called by the code

Compiled demo

- [make demo]
  - “the ‘gets’ function is dangerous and should never be used” 🤔
  - It just reads a line of text, how bad can it be?
- [readelf -S demo]
  - Flags:
    - All sections are readable (kinda useless otherwise)
    - A: allocated on program load (otherwise just metadata)
    - W: writeable
    - X: executable
  - .text: where machine code goes (NOT TEXT DATA!!!)
  - .rodata: read-only data
  - .data: read-write data
  - .bss: zeroed read-write data (only size given, not contents)
  - .symtab and .dynsym: symbols (names and addresses of functions and variables)

Compiled demo: reading stuff

- [strings demo] gives you strings in demo
- [nm demo] gives you symbols in demo
- [objdump -D demo -M intel] tries to turn everything (data or code) into assembler (disassembling), and prints it out
  - Using -d instead just targets the executable sections
  - [objdump --disassemble=bar] will disassemble only the symbol main

Hack demo

- Program is boring
  - Let’s hack it
- [xxd -p hax]
  - Bunch of As and some random-looking bytes
- [cat hax - | ./demo]
  - Types the contents of hax, then lets us interact as normal
  - Prints out weird text
  - And it’s is now a shell???

What?

- What?

What?

- Strace shows what the program asks the kernel to do for it (system calls)
  - File I/O
  - Start processes
  - Other stuff
- [cat hax - | strace ./demo]
  - Why is the process asking to be turned into a shell?

My analogy is definitely really clever and not just an excuse to hit things with a hammer

- This is how the programmer feels when the light turns off
- Or how the circuit designer feels when you whack the PCB with a hammer
- We are missing some lower-level understanding

Assembly refresher

- jmp: jumps to some point in memory
- call: jumps to some point in memory, can come back with ret
- mov: sets some memory or register value
- lea: sets a register to the address of the rhs
- All other instructions are either obvious (add, xor) or easily googlable

Lifetimes

- Memory in modern processes is in 3 groups
  - Static: allocated by loader, deallocated by reaper
  - Heap: allocated by malloc, deallocated by free
  - Stack: allocated by push, deallocated by pop
- What’s the difference?

Static

- There for the entire life of the program
  - Being allocated at the start costs basically no extra time
  - The .bss, .rodata and .data sections set this up
- This is where you put your code and global variables
  - You don’t want those suddenly disappearing
  - You can’t allocate code segments without any code!
- Can’t be allocated/deallocated
  - All functions have to agree on what memory is theirs
    - Remember the “no mutable global variables” thing?
  - Must allocate the maximum you will ever need, which is inefficient
  - Technically abuse of mmap can, but that’s another story

Heap

- The malloc and free functions control this
  - Datastructures used to control this can be messed with, but that’s another talk
- This is where you put data of unknown size, or memory that needs only needs to exist for some time
- Heap allocation has to go ask the kernel for memory during run time, so slow
  - Modern heap allocators do clever things to make this faster, but still generally quite slow
- We don’t use this in the demo program, so it can’t be the problem

Stack

- CPU Instructions `push` and `pop` control this
- [board demo]
  - Stack grows down (annoyingly)
  - Stack frames are placed after return pointer
  - FILO
- Keeps track of return pointer
  - How ret finds which call to return to
- Also can be used for local variable allocation [demo]
  - Very easy cleanup of variables, much faster than malloc
- Putting two *very* different types of data together
  - Hmm, sounds like hax

Sounds like hax

- [ret2func board demo]
- Gets writes past the end
- Easy in theory, need some tools for real-world

Finding the offset

- Gdb + gef
  - Can do with just base gdb, but effort
  - https://github.com/hugsy/gef
- How do we find this offset?
  - Trial and error (spam As until segfault and wiggle around a bit)
  - Clever maths (debruijn sequence)
- https://github.com/c3-ctf/debruijn/releases/tag/v0.0.1
- [debruijn demo into gdb]
  - [debruijn 2 AA]
  - ‘r’ starts program
  - Typing in our sequence segfaults
  - ‘i f’ prints current frame info
  - ‘x/s $rsp’ prints the string at the stack pointer
  - [debruijn 2 AA IA] gives us the offset 16 (only need n letters)

Building the payload

- I spent years trying to build payloads using python, JS and even PERL!
- nasm is best tool I’ve found
  - Can easily type in addresses, and it handles endianness
  - Can handle instructions
  - Available on basically anything that can be called a computer
- Non-assembly syntax:
  - db: byte, dw: 16-bit int, dd: 32-bit int, dq: 64-bit int
  - times <n> <instruction>: repeats <instruction> n times
  - [BITS <n>]: tells nasm to assemble for an <n> bit computer
  - We’re on amd64, so use [BITS 64] and dq for addresses
- Use nm or cutter to find interesting functions to return to
- Let’s go! [ret2func demo]

But where’s the shell :(

- We did some cool stuff, but we haven’t loaded a shell yet
- Where else can we return?
  - We can “return” to any point in a function!
  - We can even “return” outside of a function! (as long as it’s executable)
  - gef’s vmmap shows executable segments:
  - The stack is readable, writable AND executable!!!
- [nano hax.S]
  - Code looks weird, you have to write it to work anywhere

Leaving the 90s

- This was really cool back in the 90s
- Made the internet unusable
  - Haha funny dialup hax
  - Worms
  - Mostly just random destruction
  - Sometimes not just for lulz

The war begins!

- W^X
  - Why the hell can you execute the stack? actually some good reasons
  - Stop executing the stack!
- Now we can only crash the program, right?
  - I’m giving this talk, so guess the answer...

ROP

- Return Oriented Programming
  - “ret” just jumps to the next item on the stack
  - Find “gadgets” that do useful things, followed by a “ret”
    - pop <register> allows us to set registers!
  - Chain them!
- ROPgadget helps find useful things
  - Can even build whole ROP chains, but usually overly complicated
- For more advanced binaries, you want cutter
  - Great for reverse engineering and binary patching
  - https://github.com/rizinorg/cutter/releases
- [demo rop.S]
  - Cutter gives the address of all strings in the binary
  - Use the “system” function (ret2libc)

The war continues!

- The attacker is using the addresses of important things to do bad stuff
- If we randomised those addresses, it would make their lives harder!
- This is called Address Space Layout Randomisation (ASLR)
- ASLR is enabled kernel-wide on Linux
  - So that’s why we can’t just hard-code a stack address

There’s a lot more

- This only scratches the surface of what memory corruption and binary exploitation can do:
  - LD_PRELOAD and linker fun
  - Heap attacks (use-after-free, heap smashing)
  - Spooky hardware attacks
- There’s also a lot more theory behind this

Bypassing ASLR

- Statically linked libraries are all in the same section
- Relative addresses in the same section are constant
  - If you find one address in a section, you can get them all
- Most programs are not position-independent code
  - If you have enough gadgets in the non PIC section, you don’t care about ASLR
  - To interface with the PIC parts (such as so/dll files), just jump to the PLT, which will redirect you to the main function
- If not enough is in the PLT, or it’s a position-independent executable, then you need to be able to read arbitrary bits of RAM
  - This means you need 2 vulns instead of one
- Failing all of this, you have to brute force the offset, which is noisy and slow

Stack cookies

- You’re overwriting the stack
- So check if the stack has been overwritten
- Random number stored in a register, check before each ‘ret’
- Extra memory read after each function is slow
  - Sometimes only done on “dangerous” functions

Haha good luck

- You need to either:
  - Be able to skip over the stack cookie without overwriting it with ‘A’s
  - Read the stack cookie, which requires another vuln
  - Brute force the cookie
- 32-bit systems have only 24 non-zero bits of stack cookie, as the first byte is 0 to protect against string functions
  - Can be reasonable to brute force all 16 million possible values
  - Not for 64 bit though!
- Failing this, and assuming you can’t brute force it, you’re basically stuck

However, not fixed

- Optional security isn’t secure
- Other related exploits that can’t be patched in this way
- Old software still in use that isn’t secure

Bounds checking

- Just bounds check
- It’s really easy
- It’s often a single register comparison, and so has no significant overhead
- #define _GLIBCXX_DEBUG or #define _GLIBCXX_ASSERTIONS
  - Slower, but enforces proper bounds checks with C++
- #define FORTIFY_SOURCE 1
  - Should basically always be on
- Or use a cringe language like Rust

CTF challenges & extra resources

- The pwn challenges are of this form
- LiveOverflow on youtube has a great series
  - It’s what got me started
  - https://www.youtube.com/watch?v=iyAyN3GFM7A&list=PLhixgUqwRTjxglIswKp9mpkfPNfHkzyeN
- Message me on Discord
- Ask me a question IRL!
- Just practice a lot :)