Lecture 08
Code Reuse (part 1)

Computer and Network Security
November 18, 2019
Computer Science and Engineering Department

- static & dynamic analysis
- ASCII armored address space
- stack guard, canary value
- DEP: Data Execution Prevention
- ASLR: Address Space Layout Randomization

Data Execution Prevention

- writable code may not be executed
- stack, heap, data, bsss
- you cannot (easily) inject code (and run it)

Code Reuse

- use existing code
- text and library text
- interesting to use/call functions
- smaller pieces may also be used
- no need to inject code

return-to-libc

- call existing library functions
- canonical exploit is calling `system("/bin/sh")` in `libc`
- code pointer is overwriten with address of library function

Return-Oriented Programming

- use smaller pieces of code
- pieces are called *ROP gadgets*, ending in `ret` instruction
- payload consists of data on stack and pointers to ROP gadgets

High-level View

- call functions in libraries (`libc`)
- call `system("/bin/sh")` for a shell
- call `puts()` for information leak

Steps in a ret-to-libc attack

- buffer overflow required
- identify function addresses and addresses of arguments (such as the *"/bin/sh"* string)
- ovewrite code pointer with function address
- place arguments on stack

Sample Payloads for ret-to-libc attack

- `padding + overwrite_lib_address + irrelevant_ret_address + arg1 + arg2 + ...`
- `offset*"A" + p32(system_address) + 4*"B" + p32(bin_sh_address)`
- `offset*"A" + p32(write) + 4*"B" + p32(1) + p32(buf_address) + p32(buf_len)`

Protecting Against ret-to-libc Attacks

- use ASLR to randomize function addresses in library
- use stack canary
- need information leak to bypass protection mechanisms

ret-to-libc Shortcomings

- cannot chain multiple calls due to stack limitations
- function calls may be too coarse; you may need smaller chunks

Remember `ret` Instruction

- pop instruction/code pointer from stack
- code pointer was placed by `call` instruction ...
- ... or by exloit payload

`pop-ret`

- a sequence: data + code pointer on the stack is used by a `pop; ret` sequence
- e.g.: `pop eax; ret`: place data in eax and pop instruction pointer from stack
- you may use `pop2-ret` or `pop3-ret` etc.

Freeing the Stack

- chain together multiple functions
- after calling a function do a `pop-ret` or `popX-ret` to free function arguments

ROP Payloads

- `offset*"A" + p32(puts) + p32(pop_ret_gadget) + p32(puts_string_address)`
- `offset*"A" + p32(write) + p32(pop3_ret_gadget) + p32(1) + p32(buf) + p32(buf_len)`

Return Oriented Programming

- using function calls + ret-based calls to chain together code reuse chunks
- makes use of *ROP gadgets*
- is a Turing-complete *language*

- small sequences ending in `ret`
- use `ROPgadget` tool (comes with `pwntools`)
- use `ropgadget` or `ropsearch` or `asmsearch` in PEDA

- chain together function calls + ROP gadgets
- do information leak, rewrites, open sockets, run shells

- leak information: variables, addresses
- open shell
- call `mprotect()` to disable DEP and then inject shellcode

- DEP
- code reuse
- return-to-libc
- ROP
- ROP gadget
- ROP chain
- `ROPgadget`

- `https://www.blackhat.com/presentations/bh-usa-08/Shacham/BH_US_08_Shacham_Return_Oriented_Programming.pdf`
- `https://trailofbits.files.wordpress.com/2010/04/practical-rop.pdf`
- `http://codearcana.com/posts/2013/05/28/introduction-to-return-oriented-programming-rop.html`
- `https://github.com/JonathanSalwan/ROPgadget`