Worm enabling exploits

Cyber Security Lab
Spring ‘10

Background reading

- Worm Anatomy and Model
  - http://portal.acm.org/citation.cfm?id=948196
- Smashing the Stack for Fun and Profit
  - http://www.phrack.com/issues.html?issue=49&
- The Shellcoder's Handbook
  - At the library

More Reading

- Steve Hanna's Shellcode page
  - http://vividmachines.com/shellcode/shellcode.html
- Once Upon a Free()
  - http://www.phrack.org/issues.html?issue=57&id

Outline

- Review worm structure
- Examine exploited vulnerabilities
  - Buffer Overflow
  - Return to Libc
  - Format String exploits
  - Heap Overflow

What is a Worm?

- An autonomous process that can cause a copy of itself (or a variant) to execute on a remote machine.
- Various Goals
  - Install trojan's for later access
  - Install zombies for later DDoS or other activities
  - Install spies for information gathering
  - Personal fame
- Generally varies from a virus in that it propagates independently.
  - A virus needs a host program to propagate.
  - But otherwise, many of the issues between worms and virus are the same

Life Cycle of a Worm

- Initialization:
  - Install software, understand the local machine configuration
- Payload Activation:
  - Activate the worm on the current host
- Network Propagation:
  - Identify new targets and propagate itself
  - The cycle starts all over on the newly infected devices

Network Propagation in More Detail

- Target Acquisition: Identify hosts to attack.
  - Random address scans (Code Red) or locality biased (Nimda)
  - Code Red v2 effectiveness changed based on good seeding
- Network Reconnaissance: Determine if the target is available and what is running on it
- Attack: Attempt to gain root access on the target
  - Traditionally this has been buffer overflow
  - Can also attack other weaknesses like weak passwords
- Infection: Leverage root access to start the Initialization phase on the new host

Example Worm: LION

- Active around 2001
- Three versions
- Not a particularly effective worm
  - Uses a BIND exploit that attacks the “named” daemon
    - Not activated on default RedHat 6.2 installations
    - Administrator would have to explicitly add to inetd table and run as root
- Variant of the earlier worms
  - ADMworm, Millenium Worm, Ramen worm

Lion Life Cycle

- Attempts connection to TCP port 53 on candidate target hosts
  - Selects random class B network blocks to scan
- If target responds, send malformed UDP IQUERY packet to UDP port 53
  - Used to determine if target is running vulnerable version of Linux running BIND 8
- If vulnerable, send overflow packet
  - Attack code walks file descriptor table of exploited process to find FD of initial TCP connection
  - Duplicates FD to stdin, stdout, stderr
  - Spawn /bin/sh running at root

Lion Life Cycle Continued

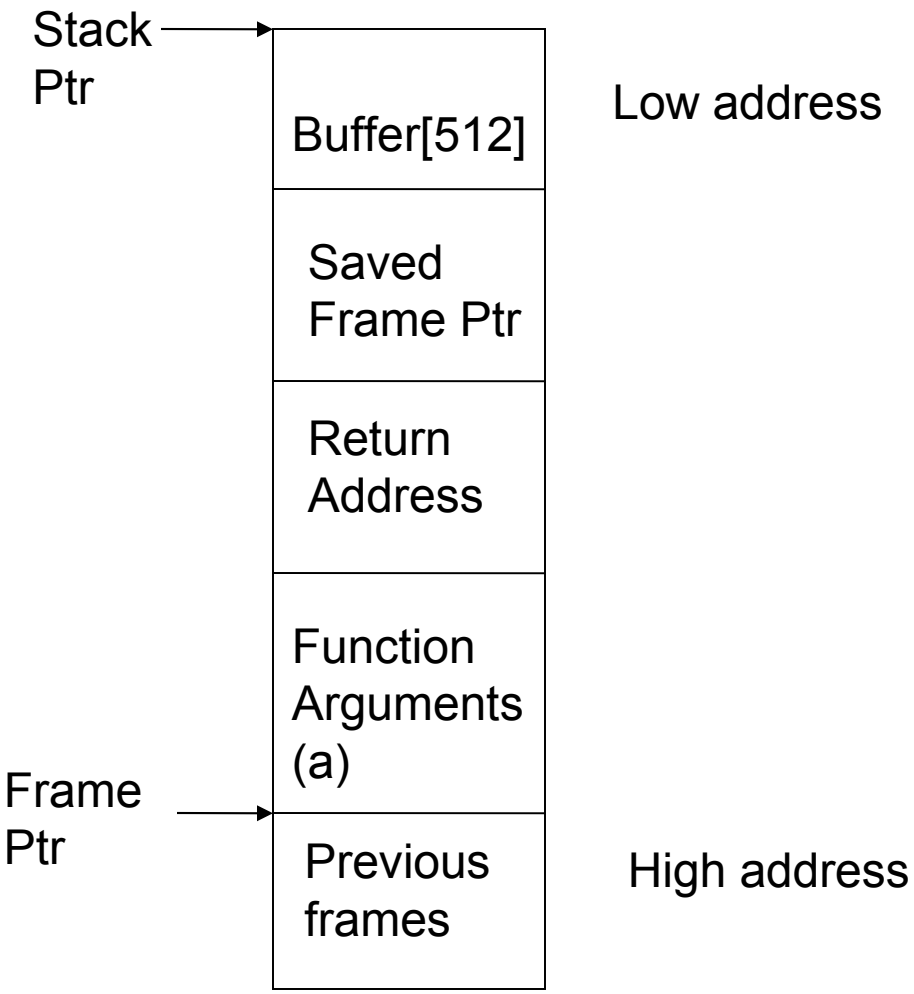
- Now can use original TCP connection as control channel to send shell commands
  - Download and install software
    - Versions 1 and 2 download from fixed site
    - Version 3 uses Ramen distribution code to download from infecting host
  - Send password files to central location for later analysis
  - Cover tracks. Erase logs and temporary files

Buffer Overflow Exploits

- Write too much data into a stack buffer
  - Replace return address on the stack with address of attack code
  - Generally attack code attempts to start a shell
    - If process is SetUID root, shell will be root
    - Attack code is often in the buffer

Stack Structure

```
void func(char *a) {
   char buffer[512];
   strcpy(buffer, a);
   ….
 }
```

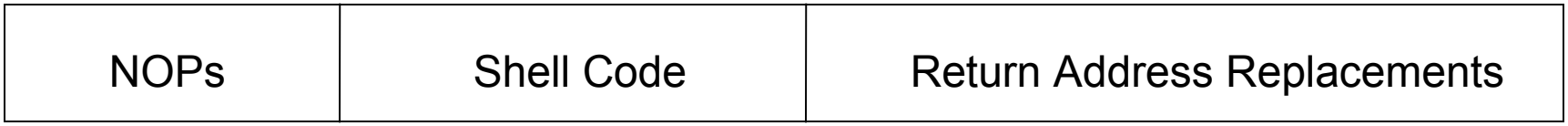


Shell Code

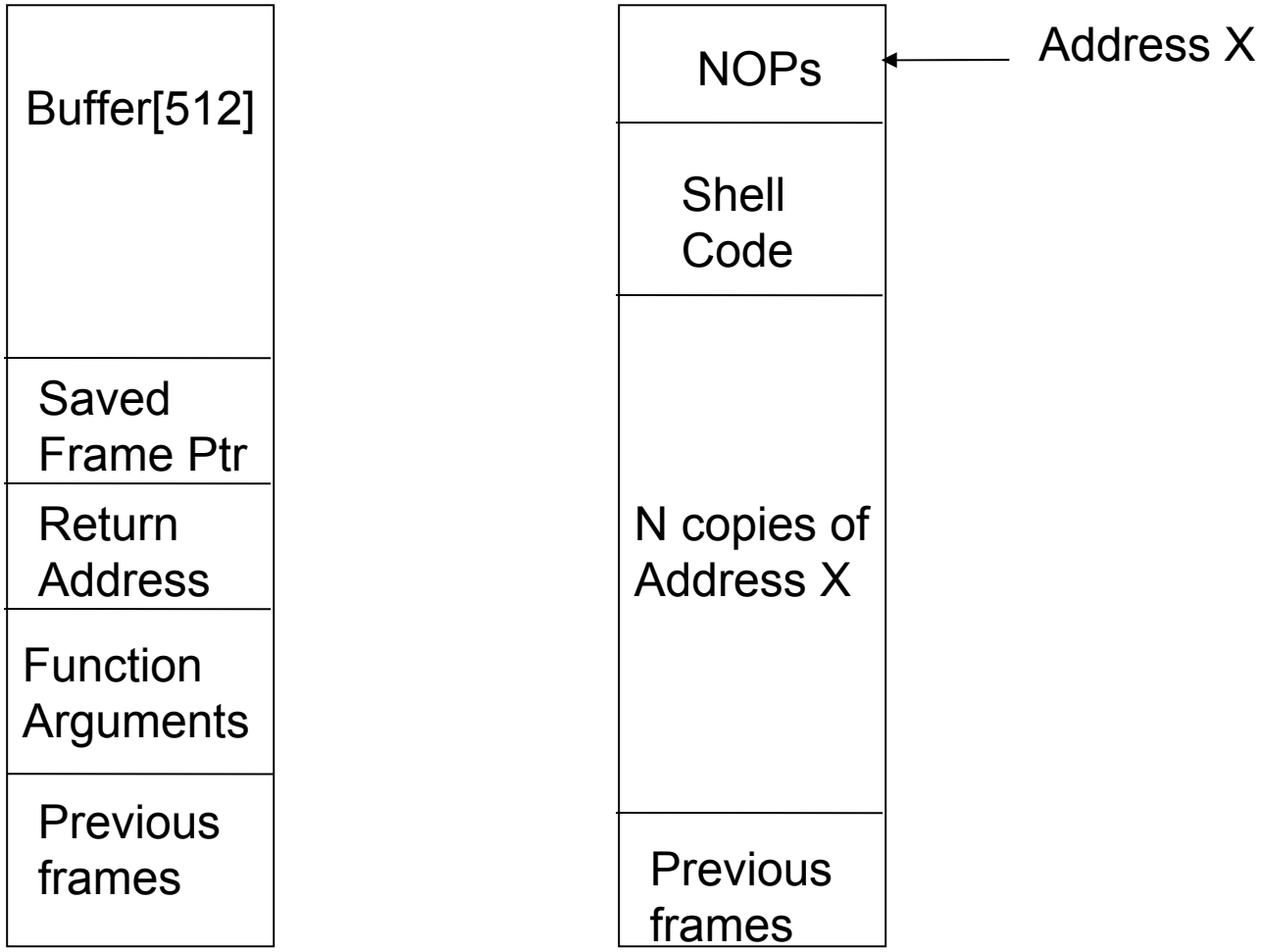
- Insert code to spawn a shell
- Phrack article discusses how to do this from first principles
  - Create assembly code to exec /bin/sh
  - Use GDB to get hex of machine code
  - Rework assembly as necessary to avoid internal 0's
    - Could break attack if strcpy is used by attack target
- Will result in a hex string like:
  - "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh"

Structure of Buffer

- Buffer more than 512 bytes will replace other information on the stack (like return address)
- Problem is determining absolute address in buffer to jump to and ensuring you replace the return address
  - Pad with leading NOPs and trailing return addresses
  - Then your guesses on the stack structure do not need to be exact



Copied Stack



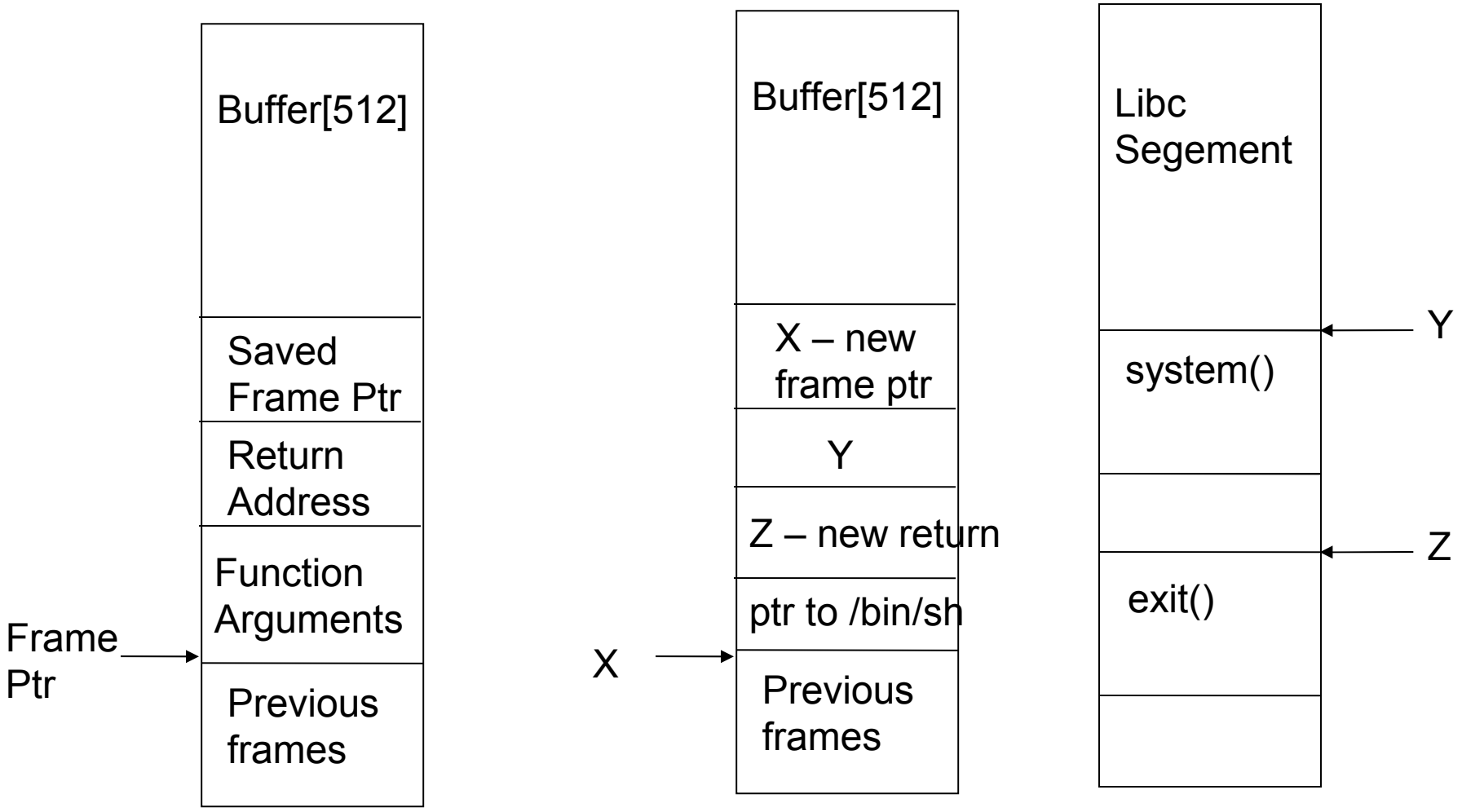
Calculating New Return Address

- If you have source
  - Use GDB to find stack address at appropriate invocation
    - GDB reporting may not be accurate, might take several guesses
  - Use Eggshell program
    - Approximate target program
    - Takes buffer size and offset arguments
    - Computes candidate buffers
    - Emits buffers in environment variable named EGG
    - Creates new shell on the way out so EGG is available after program has completed
- If you don’t have source
  - Brute force?
  - Examination of core files or other dumps

Return to libc

- Make stack non-executable to protect from buffer overflow
  - Newer windows feature
  - Feature in some flavors of Unix/Linux
- Adapt by setting the return address to a known library
  - Libc is home to nice functions like system, which we can use to spawn a shell.

Return to Libc Stack



Protections

- No execute bit
- Address space randomization
- Canaries
- Use type safe languages
- Avoid known bad libraries

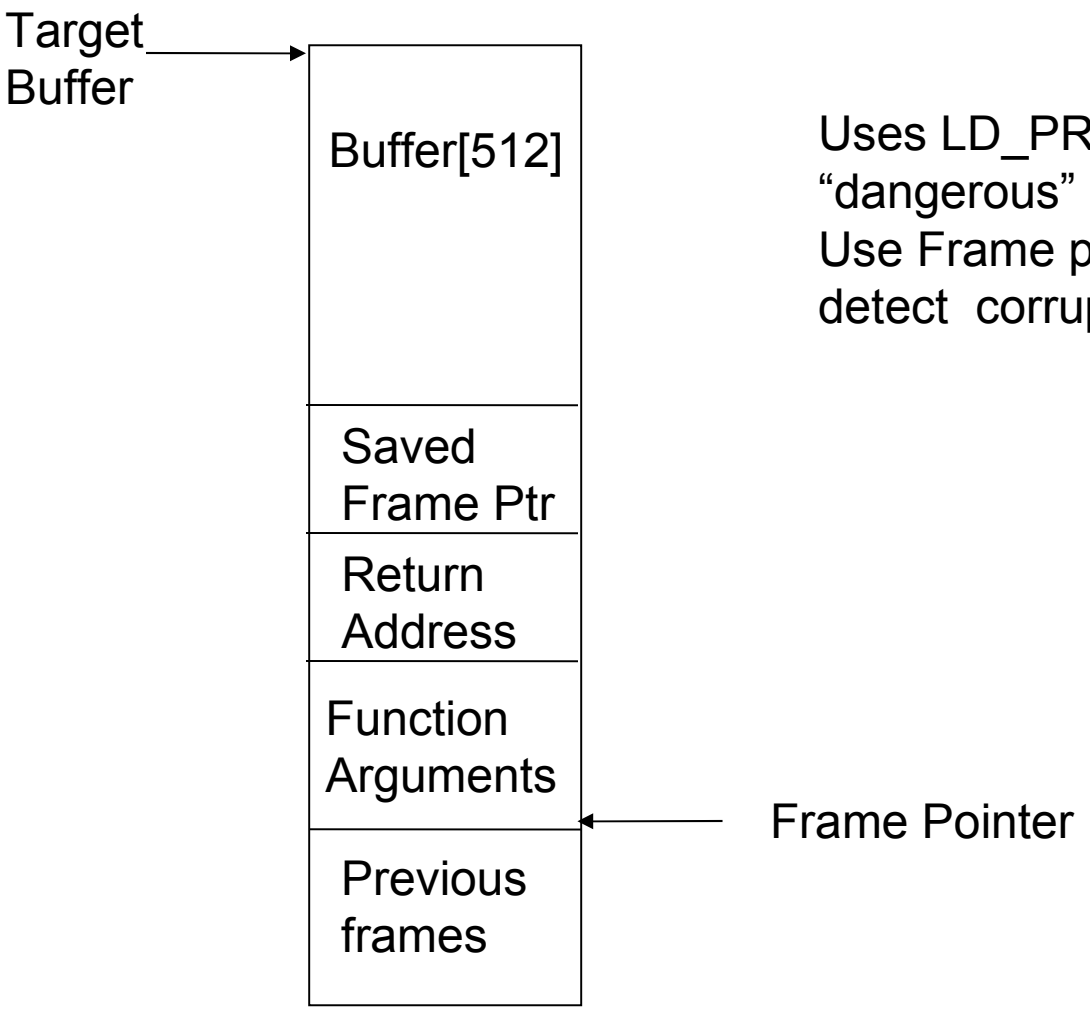
Address Space Randomization

- Vary the base stack address with each execution
  - Stack smashing must have absolute address to over write function return address
  - Enabled by default in some linuxes (e.g., FC3)
- Wastes some address space
  - Less of an issue once we have 64 bit address space
- Not absolute
  - Try many times and get lucky
- Does not help return to libc or heap overflows

Tools for Buffer Overflow Protection

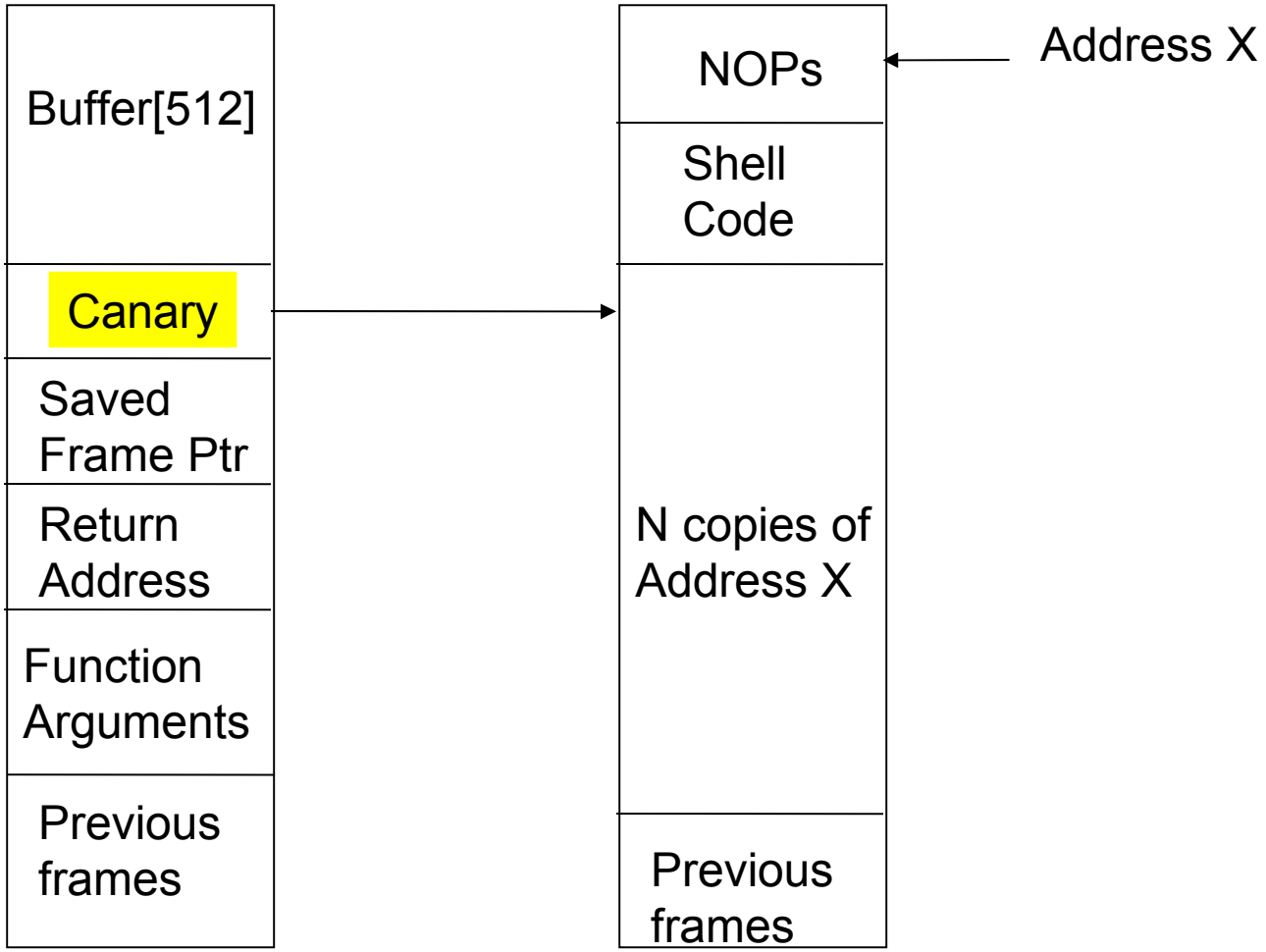
- LibSafe
  - http://www.research.avayalabs.com/project/libsafe/
  - Intercept calls to functions with known problems and perform extra checks
  - Source is not necessary
- StackGuard and SSP/ProPolice
  - Place “canary” values at key places on stack
    - http://en.wikipedia.org/wiki/Stack-smashing_protect
  - Terminator (fixed) or random values
  - ProPolice patch to gcc

LibSafe



Uses LD_PRELOAD to intercept all "dangerous" calls.
Use Frame pointer and buffer address to detect corruption of stack

Canary Values



Non-Executable Stack

- Set page as non-executable
  - Supported by newer AMD and x86 chips
  - Supported by some OS's
- Does not protect against return to libc or heap attacks.

Format String Errors

- What is a format string?
  - printf(“Foo 0x%x %d\n”, addr, count);
- What happens if the arguments are missing?
  - printf(“Foo 0x%x, %d\n”);
- What if the end user can specify his own format string?
  - printf(fmtstring)

Information Disclosure

- By specifying arbitrary %x's (or %d's) you can read the stack
  - Made easier by direct parameter access
  - "%128\$x" – print the 128'th argument as a hex
- Looking at the stack you can see the address to your own format string

Reading arbitrary addresses

- You can load an address into the first 4 bytes of your format string
- If you know the offset of the format string on the stack, use %s to read the string starting at that address
  - formatstr = $'\x55\x4d\x06\x08%272$s';
  - printf(formatstr)
- So, we leak information, but printf is read only, right?

Writing data with printf

- The %n parameter writes the number of bytes written so far by printf to the corresponding int * pointer
- Kind of awkward, but does enable the dedicated fiddler to write arbitrary data at arbitrary locations
  - Only writes one byte at a time
- Likely targets
  - Return addresses
  - Data, like terminating passwords we are checking
  - Global Offset Table (GOT) – library function pointer table

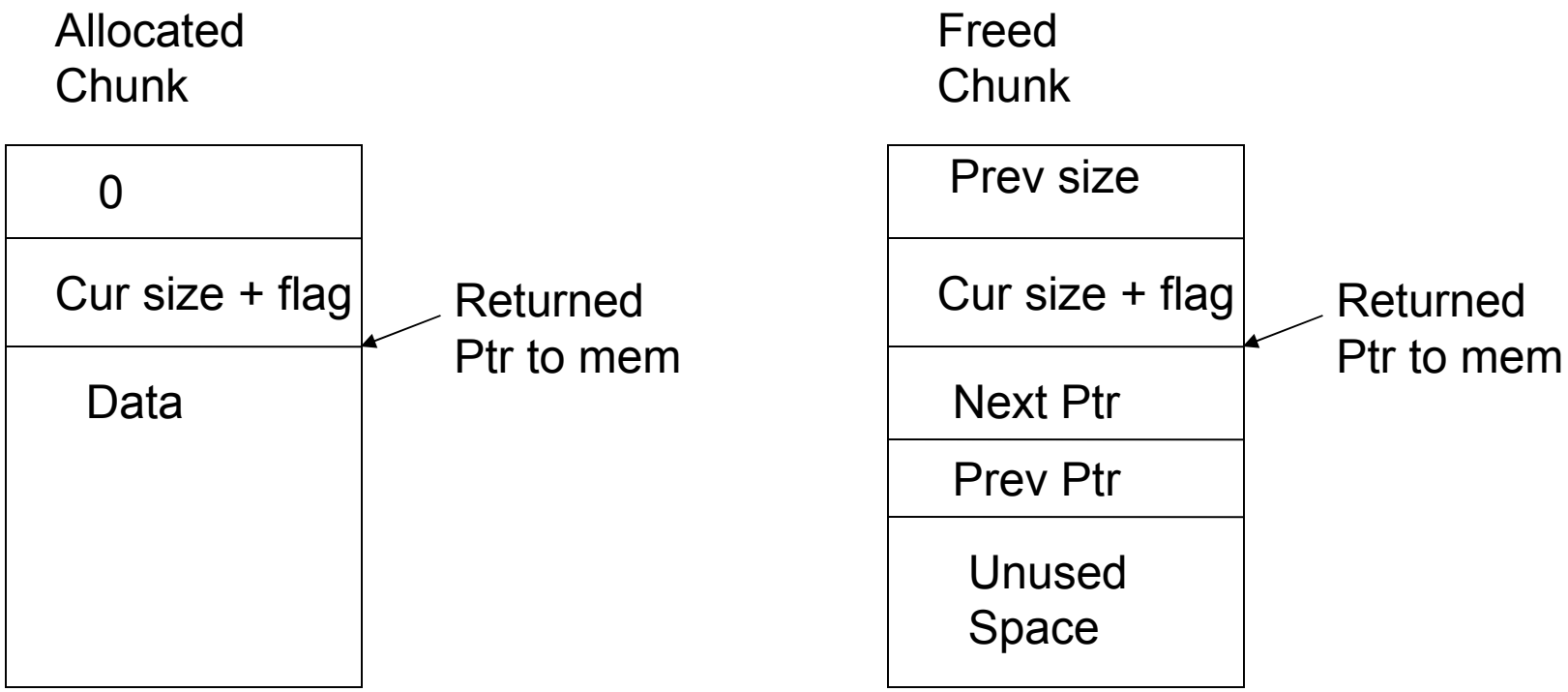
Format string errors easily avoided

- Never accept raw format strings from end user
  - Never allow
    - printf(buf)
  - Instead do
    - printf(“%s”, buf);

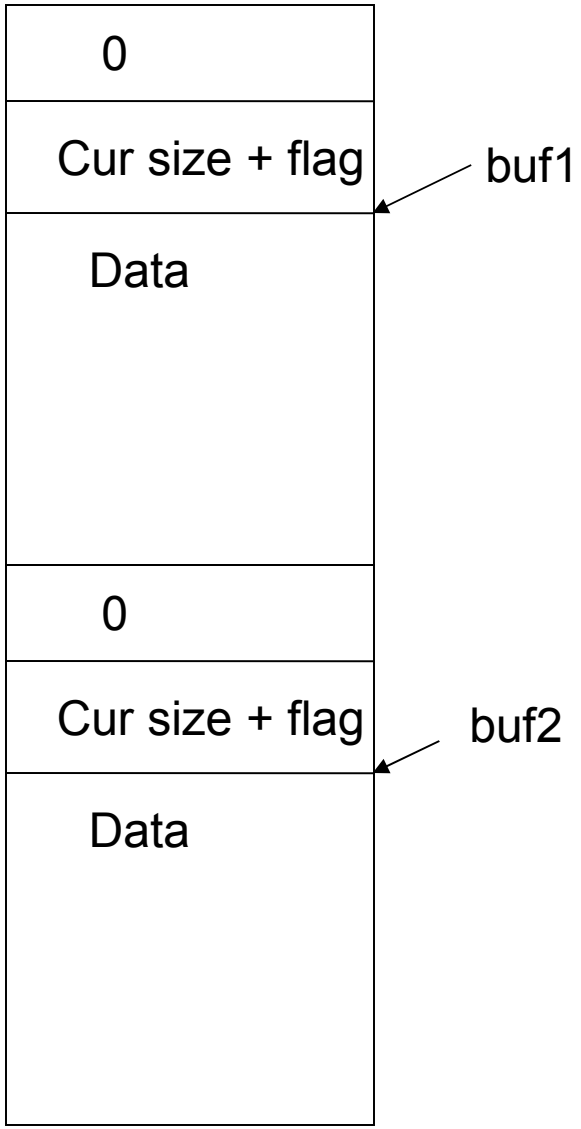
Heap overflows

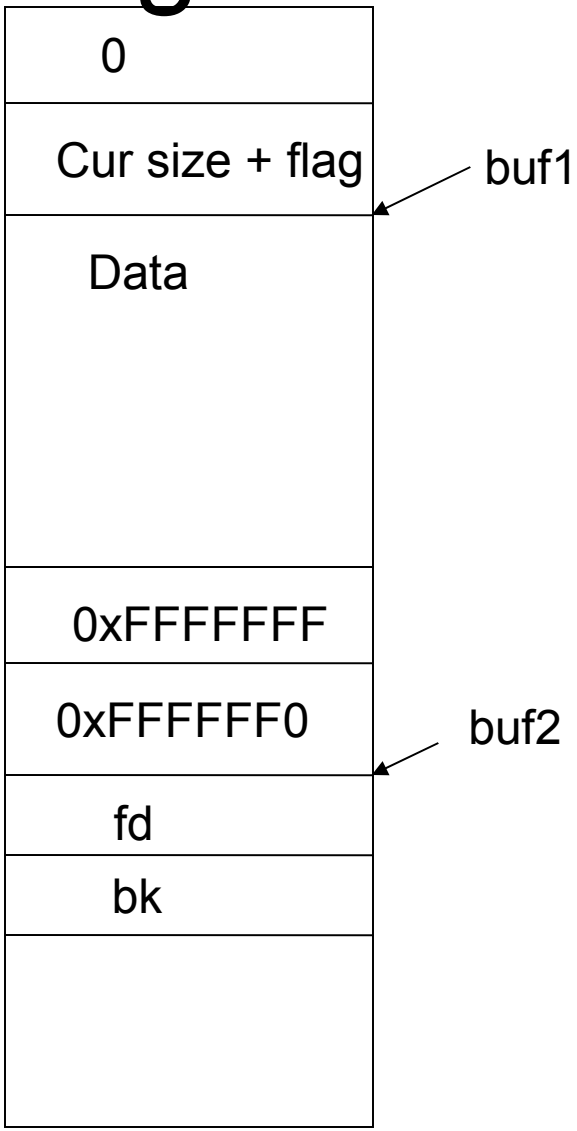
- Gain control by overflowing heap allocated buffer
- Heap imposes additional structure on large blocks of memory given by OS
- Control structures intermingled with user data in heap memory
  - Specific attacks very dependent on details of particular malloc implementation

Example Structure



Control Memory Through Free

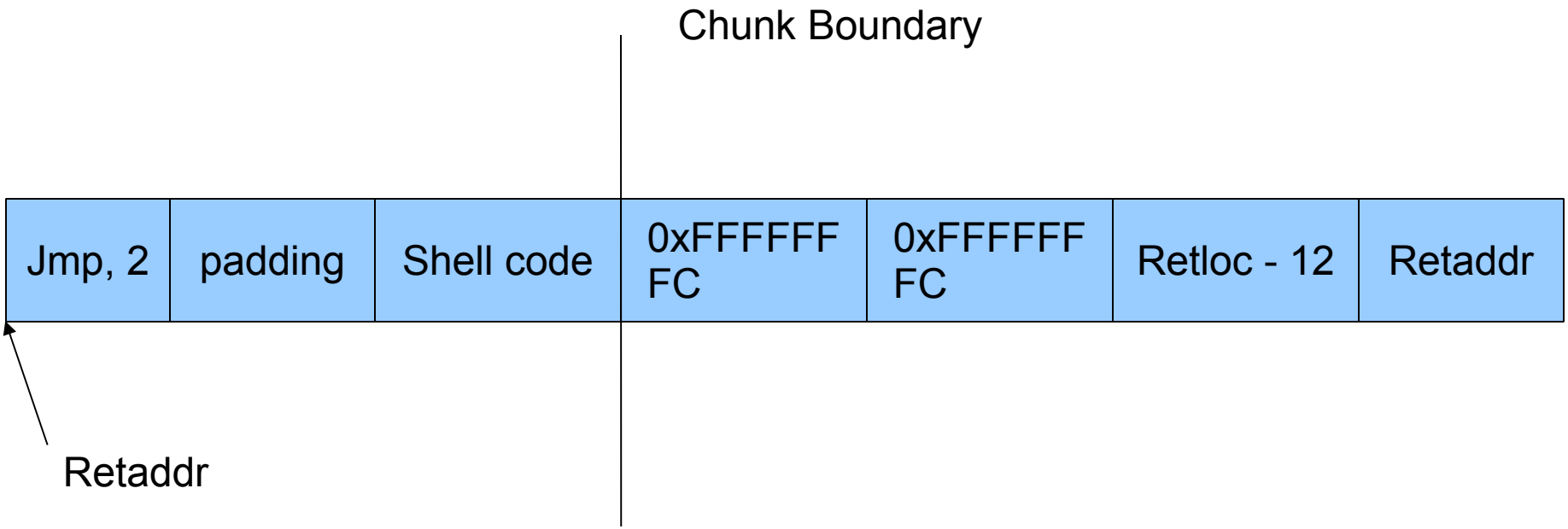




Exploiting Heap Control Structure

- Overwrite into the next “free” block
- Set or unset low bit of size to control path through free
  - Unlink will use the first two words in the memory to remove itself from linked list.
  - You can put any memory address there, e.g. Stack return location, and control broader execution flow.

Poison buffer



Heap attack protections

- Randomization could help use here too.
  - DieHard (DH) Memory Allocator
  - http://prisms.cs.umass.edu/emery/index.php?pa

Summary

- Worms rely on exploits of networked services
  - Goal: get a shell started at high privilege
  - Even shell at low privilege gives attacker a foothold to attack locally
- Exploits need to write specific data and specific addresses
  - Trick data structures
  - Use mechanisms in unexpected ways