CS4459.003
Cyber Attacks & Defense Lab

Shellcoding Part 1

Feb 13, 2024

From Last Class

- Buffer overflow attacks
  - Calling convention + stack layout
  - No bound check to guard the boundaries

Learning Objectives

- Writing programs in assembly (GAS/gas)
- Shellcode/Shellcoding: Load your own payload
    - Your payload ‘`get_a_shell()`’
    - Linux access control
- System call
    - Vs. library call (glibc)
    - 32-bit vs. 64-bit

Buffer Overflow

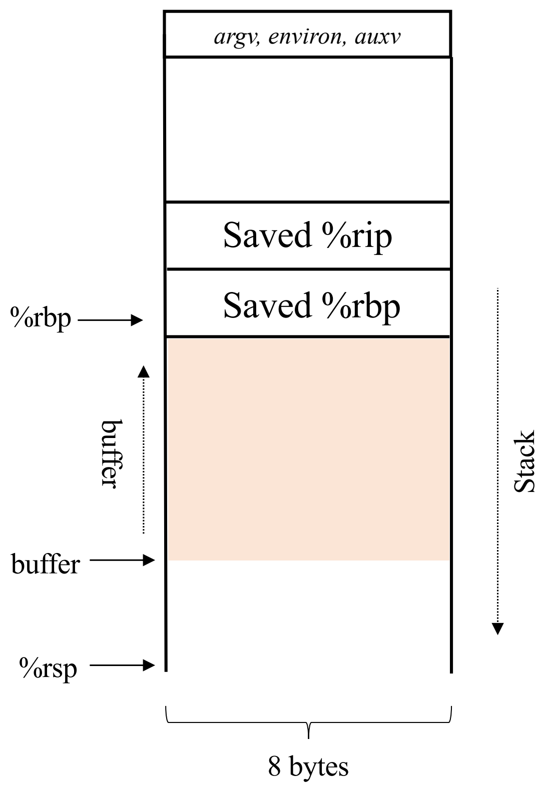
- Overflow buffer and overwrite
  - local variables
  - Previous %EBP
  - Function’s return address

```
main () → run() → recv_input();
```

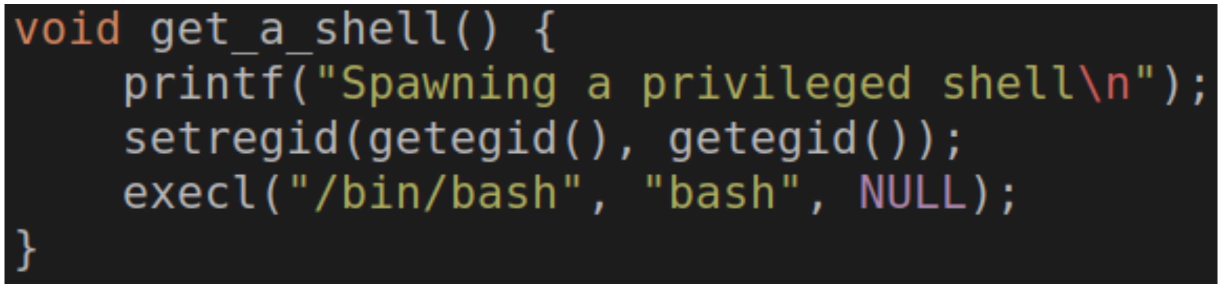
- Jump to where you wish to run

```
get_a_shell();
```

64-bit Stack

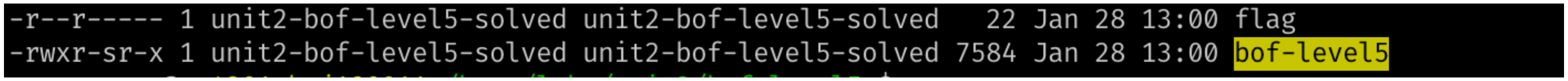


get_a_shell()



```c
void get_a_shell() {
    printf("Spawning a privileged shell\n");
    setregid(getegid(), getegid());
    execl("/bin/bash", "bash", NULL);
}
```

- Inherit current privilege and then execute a shell
- You can read the flag!



```
-r--r----- 1 unit2-bof-level5-solved unit2-bof-level5-solved   22 Jan 28 13:00 flag
-rwxr-sr-x 1 unit2-bof-level5-solved unit2-bof-level5-solved 7584 Jan 28 13:00 bof-level5
```

```
setregid(getegid(), getegid())
execl("/bin/bash", "bash", 0);
```

`get_a_shell(): setregid()`

`getegid()`

- Get effective GID

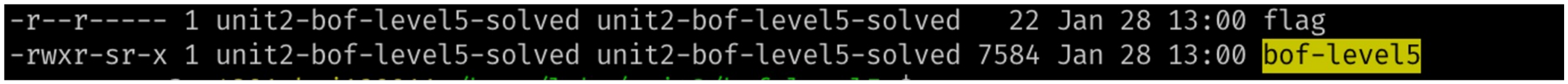
`setregid(gid_t rgid, gid_t egid)`

- Set *real* and *effective* gid

`setregid(getegid(), getegid())`

- Set real and effective *gid* as current effective *gid*
- Privilege escalation!
- Set your *gid* to unit3-level0-…

Linux Access Control: Resource Ownership



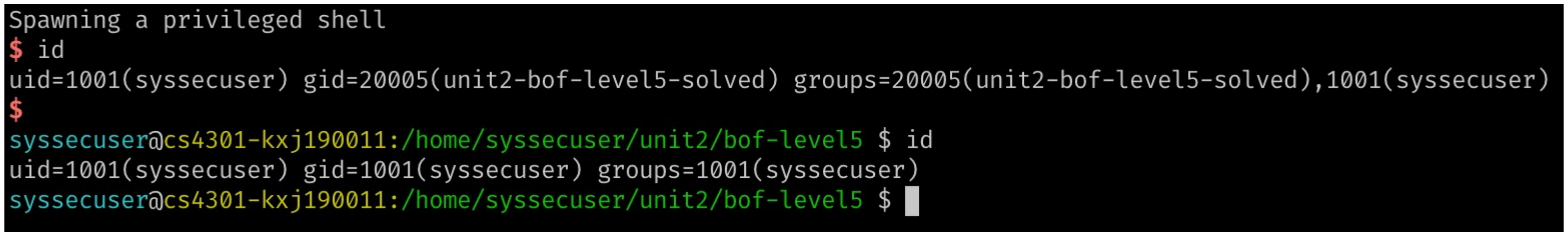
```
-r--r----- 1 unit2-bof-level5-solved unit2-bof-level5-solved   22 Jan 28 13:00 flag
-rwxr-sr-x 1 unit2-bof-level5-solved unit2-bof-level5-solved 7584 Jan 28 13:00 bof-level5
```

- Ownership for user and group
  - Who (or which group) owns the file?

- Permissions for **U**ser, **G**roup, **O**ther
  - **W**rite, **R**ead, e**X**ecution

- Program with `setuid, setguid` bits
  - Atop the privileges of the user, run with the privileges of the file owner instead

Linux Access Control: Process Identity



```
Spawning a privileged shell
$ id
uid=1001(syssecuser) gid=20005(unit2-bof-level5-solved) groups=20005(unit2-bof-level5-solved),1001(syssecuser)
$
syssecuser@cs4301-kxj190011:/home/syssecuser/unit2/bof-level5 $ id
uid=1001(syssecuser) gid=1001(syssecuser) groups=1001(syssecuser)
syssecuser@cs4301-kxj190011:/home/syssecuser/unit2/bof-level5 $
```

- UserID (UID) and GroupID (GID)
  - Defined from `/etc/passwd` and `/etc/group`

- Real ID, Effective ID, (Saved ID)
  - Real ID: ID of the user that has started the process
  - Effective ID: ID the process is running with

get_a_shell()

```
execl(“/bin/bash”, “bash”, 0);
```

- *Transform* the process and *run* “`/bin/bash`” with arg0 as ‘bash’

exec* function family

`execl(filepath, “arg0”, “arg1”, “arg2”, …, “argN”, 0)`

- Run program at filepath with args… (arg list ends with ‘\0’)
- exec’**l**’ → ‘**l**’ist..

`execv(filepath, argv[]);`

argv[0] = arg0, argv[1] = arg1, …, argv[N] = argN, argv[N+1] = 0 (ends with \0)

- exec’**v**’ → **‘v**’ector

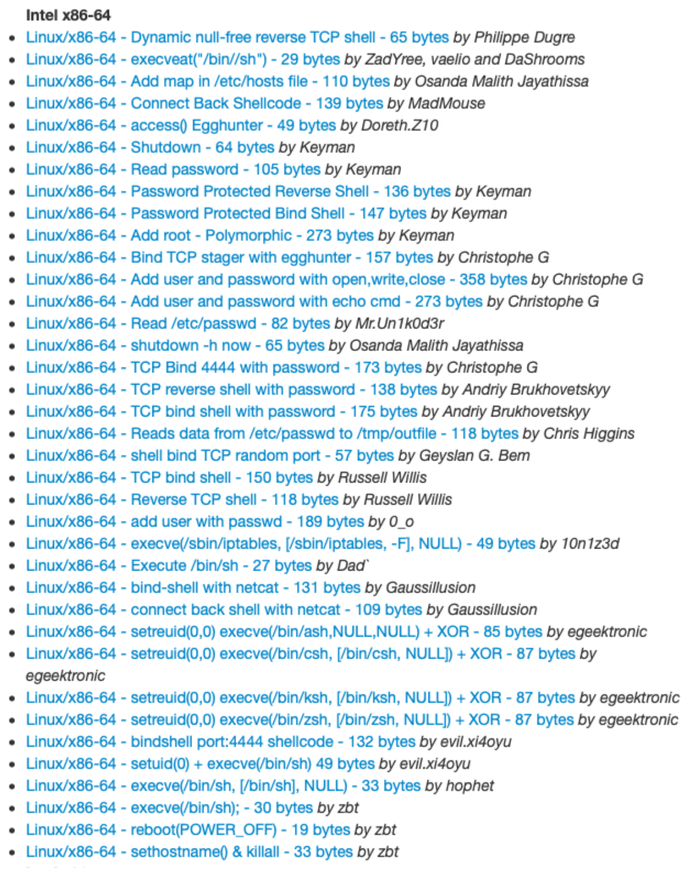
`execve(filepath, argv[], envp[]);`

- In addition to execv (for argv),
- envp[0] = env0, envp[1] = env1, envp[2] = env2, …, envp[N] = envN, envp[n+1] = 0

Shellcode

- No longer ‘`get_a_shell()`’ in real attacks
- Shellcode
  - Assembly code snippet that runs a shell (or more attacks)
- We need to have

```
setregid(getegid(), getegid());
execve(“/bin/sh”, 0, 0);
```



Intel x86-64

- Linux/x86-64 - Dynamic null-free reverse TCP shell - 65 bytes *by Philippe Dugre*
- Linux/x86-64 - execveat("/bin//sh") - 29 bytes *by ZadYree, vaelio and DaShrooms*
- Linux/x86-64 - Add map in /etc/hosts file - 110 bytes *by Osanda Malith Jayathissa*
- Linux/x86-64 - Connect Back Shellcode - 139 bytes *by MadMouse*
- Linux/x86-64 - access() Egghunter - 49 bytes *by Doreth.Z10*
- Linux/x86-64 - Shutdown - 64 bytes *by Keyman*
- Linux/x86-64 - Read password - 105 bytes *by Keyman*
- Linux/x86-64 - Password Protected Reverse Shell - 136 bytes *by Keyman*
- Linux/x86-64 - Password Protected Bind Shell - 147 bytes *by Keyman*
- Linux/x86-64 - Add root - Polymorphic - 273 bytes *by Keyman*
- Linux/x86-64 - Bind TCP stager with egghunter - 157 bytes *by Christophe G*
- Linux/x86-64 - Add user and password with open,write,close - 358 bytes *by Christophe G*
- Linux/x86-64 - Add user and password with echo cmd - 273 bytes *by Christophe G*
- Linux/x86-64 - Read /etc/passwd - 82 bytes *by Mr.Un1k0d3r*
- Linux/x86-64 - shutdown -h now - 65 bytes *by Osanda Malith Jayathissa*
- Linux/x86-64 - TCP Bind 4444 with password - 173 bytes *by Christophe G*
- Linux/x86-64 - TCP reverse shell with password - 138 bytes *by Andriy Brukhovetskyy*
- Linux/x86-64 - TCP bind shell with password - 175 bytes *by Andriy Brukhovetskyy*
- Linux/x86-64 - Reads data from /etc/passwd to /tmp/outfile - 118 bytes *by Chris Higgins*
- Linux/x86-64 - shell bind TCP random port - 57 bytes *by Geyslan G. Bem*
- Linux/x86-64 - TCP bind shell - 150 bytes *by Russell Willis*
- Linux/x86-64 - Reverse TCP shell - 118 bytes *by Russell Willis*
- Linux/x86-64 - add user with passwd - 189 bytes *by 0_o*
- Linux/x86-64 - execve(/sbin/iptables, [/sbin/iptables, -F], NULL) - 49 bytes *by 10n1z3d*
- Linux/x86-64 - Execute /bin/sh - 27 bytes *by Dad`*
- Linux/x86-64 - bind-shell with netcat - 131 bytes *by Gaussillusion*
- Linux/x86-64 - connect back shell with netcat - 109 bytes *by Gaussillusion*
- Linux/x86-64 - setreuid(0,0) execve(/bin/ash,NULL,NULL) + XOR - 85 bytes *by egeektronic*
- Linux/x86-64 - setreuid(0,0) execve(/bin/csh, [/bin/csh, NULL]) + XOR - 87 bytes *by egeektronic*
- Linux/x86-64 - setreuid(0,0) execve(/bin/ksh, [/bin/ksh, NULL]) + XOR - 87 bytes *by egeektronic*
- Linux/x86-64 - setreuid(0,0) execve(/bin/zsh, [/bin/zsh, NULL]) + XOR - 87 bytes *by egeektronic*
- Linux/x86-64 - bindshell port:4444 shellcode - 132 bytes *by evil.xi4oyu*
- Linux/x86-64 - setuid(0) + execve(/bin/sh) 49 bytes *by evil.xi4oyu*
- Linux/x86-64 - execve(/bin/sh, [/bin/sh], NULL) - 33 bytes *by hophet*
- Linux/x86-64 - execve(/bin/sh); - 30 bytes *by zbt*
- Linux/x86-64 - reboot(POWER_OFF) - 19 bytes *by zbt*
- Linux/x86-64 - sethostname() & killall - 33 bytes *by zbt*

http://shell-storm.org/shellcode/

How to Launch Shellcode?

1. Land your shellcode in the target program’s address space
    - As a part of your *input*
    - As program’s *arguments*
    - As program’s *environmental* variables
    - As the *program’s name* (argv[0])

2. Set the *return* address to your *shellcode*

3. Run

```
    setregid(getegid(), getegid())
    execve(“/bin/sh”, 0, 0);
```

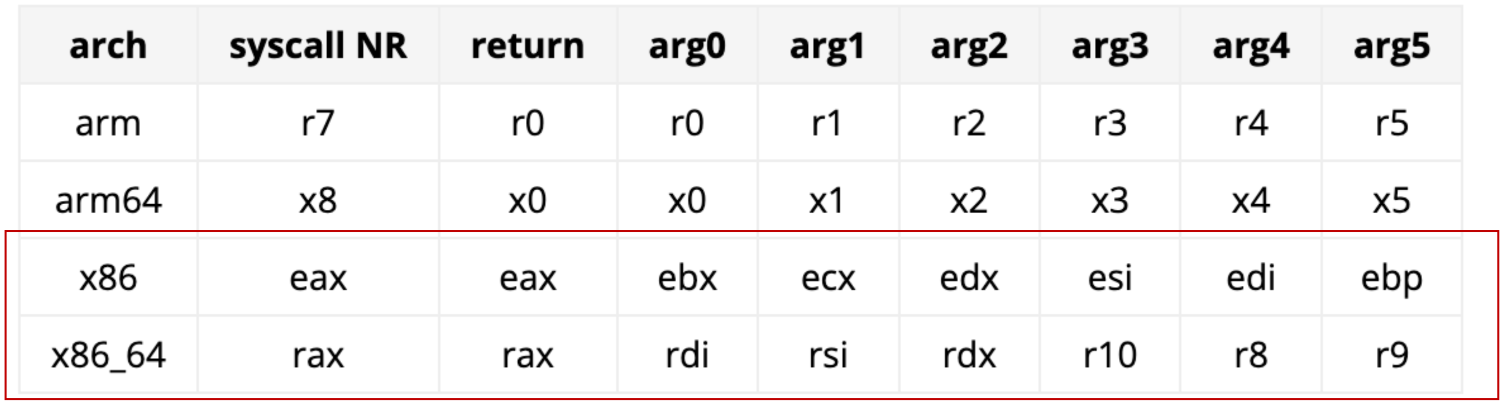
Writing Shellcode: System Call

- System call
  - Channel to talk to OS kernel
    - A function call to OS kernel
    - Context switch (expensive)

- Eventual gateway to access system resources
  - File I/O, network I/O, memory allocation
  - Set/get permissions, run program
  - Many more

- Varies for different systems and architectures
- Check system call number table for 32 bit (x86) and 64 bit (AMD64)

System Call Calling Convention



arch	syscall NR	return	arg0	arg1	arg2	arg3	arg4	arg5
arm	r7	r0	r0	r1	r2	r3	r4	r5
arm64	x8	x0	x0	x1	x2	x3	x4	x5
x86	eax	eax	ebx	ecx	edx	esi	edi	ebp
x86_64	rax	rax	rdi	rsi	rdx	r10	r8	r9

- You can see it as a calling convention between user-land and kernel-land

Invoking getuid(): x86

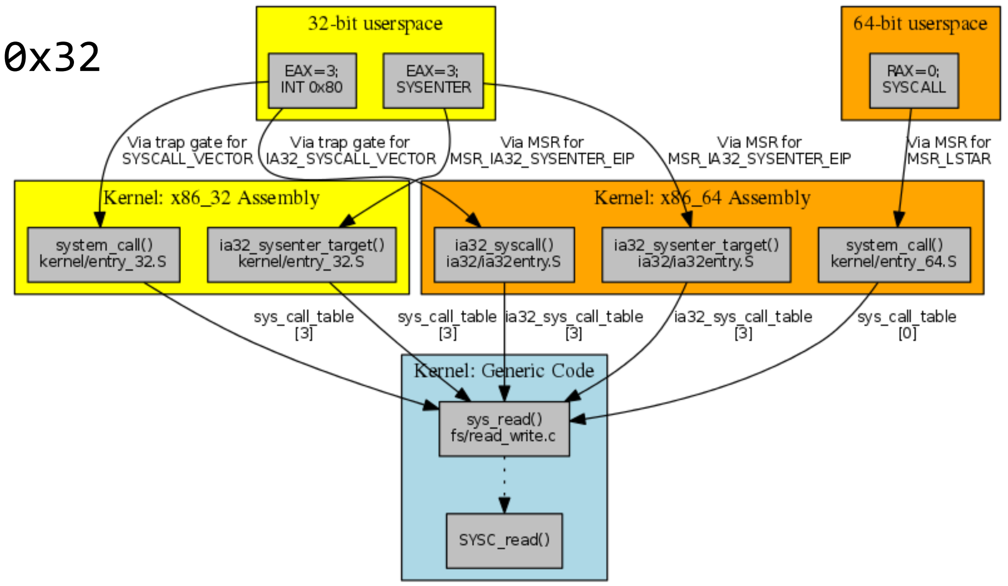
- Set %eax as target system call number

```
mov $SYS_getegid, %eax   // 0x32
```

- Set arguments
  - 1st arg: %ebx
  - 2nd arg: %ecx
  - 3rd arg: %edx
  - 4th arg: %esi
  - 5th arg: %edi
- Run

```
int $0x80
```

32-bit way


32-bit userspace
EAX=3; INT 0x80
EAX=3; SYSENTER
64-bit userspace
RAX=0; SYSCALL
Via trap gate for SYSCALL_VECTOR
Via trap gate for IA32_SYSCALL_VECTOR
Via MSR for MSR_IA32_SYSENTER_EIP
Via MSR for MSR_IA32_SYSENTER_EIP
Via MSR for MSR_LSTAR
Kernel: x86_32 Assembly
system_call() kernel/entry_32.S
ia32_sysenter_target() kernel/entry_32.S
Kernel: x86_64 Assembly
ia32_syscall() ia32/ia32entry.S
ia32_sysenter_target() ia32/ia32entry.S
system_call() kernel/entry_64.S
sys_call_table [3]
sys_call_table [3]
ia32_sys_call_table [3]
ia32_sys_call_table [3]
sys_call_table [0]
Kernel: Generic Code
sys_read() fs/read_write.c
SYSC_read()

Invoking getuid(): AMD64

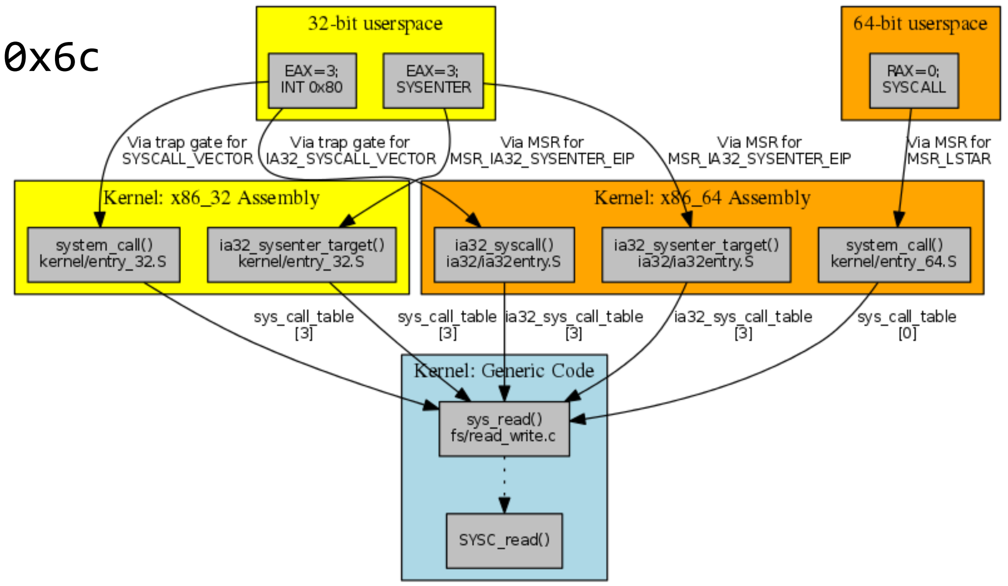
- Set `%rax` as target system call number

```
mov $SYS_getegid, %rax   // 0x6c
```

- Set arguments
  - 1st arg: `%rdi`
  - 2nd arg: `%rsi`
  - 3rd arg: `%rdx`
  - 4th arg: `%r10`
  - 5th arg: `%r8`
- Run

```
syscall
```

64-bit way



32-bit userspace
EAX=3; INT 0x80
EAX=3; SYSENTER
64-bit userspace
RAX=0; SYSCALL
Via trap gate for SYSCALL_VECTOR
Via trap gate for IA32_SYSCALL_VECTOR
Via MSR for MSR_IA32_SYSENTER_EIP
Via MSR for MSR_IA32_SYSENTER_EIP
Via MSR for MSR_LSTAR
Kernel: x86_32 Assembly
system_call() kernel/entry_32.S
ia32_sysenter_target() kernel/entry_32.S
Kernel: x86_64 Assembly
ia32_syscall() ia32/ia32entry.S
ia32_sysenter_target() ia32/ia32entry.S
system_call() kernel/entry_64.S
sys_call_table [3]
sys_call_table [3]
ia32_sys_call_table [3]
ia32_sys_call_table [3]
sys_call_table [0]
Kernel: Generic Code
sys_read() fs/read_write.c
SYSC_read()

getegid(), setregid() : x86

- Return value will be stored in `%eax`

```
mov $SYS_getegid, %eax
int $0x80
```

  - `%eax` will hold the return value of `getegid()`

- How to run `setregid(getegid(), getegid())`?

```
mov %eax, %ebx              // 1st arg
mov %eax, %ecx              // 2nd arg
mov $SYS_setregid, %eax     // syscall number
int $0x80
```

Calling **EXECVE()**

```
// execve(char* filepath, char** argv, char** envp)
execve(“/bin/sh”, NULL, NULL);
```

```
%eax = $SYS_execve
%ebx = address of “/bin/sh”
%ecx = 0
%edx = 0
int $0x80
```

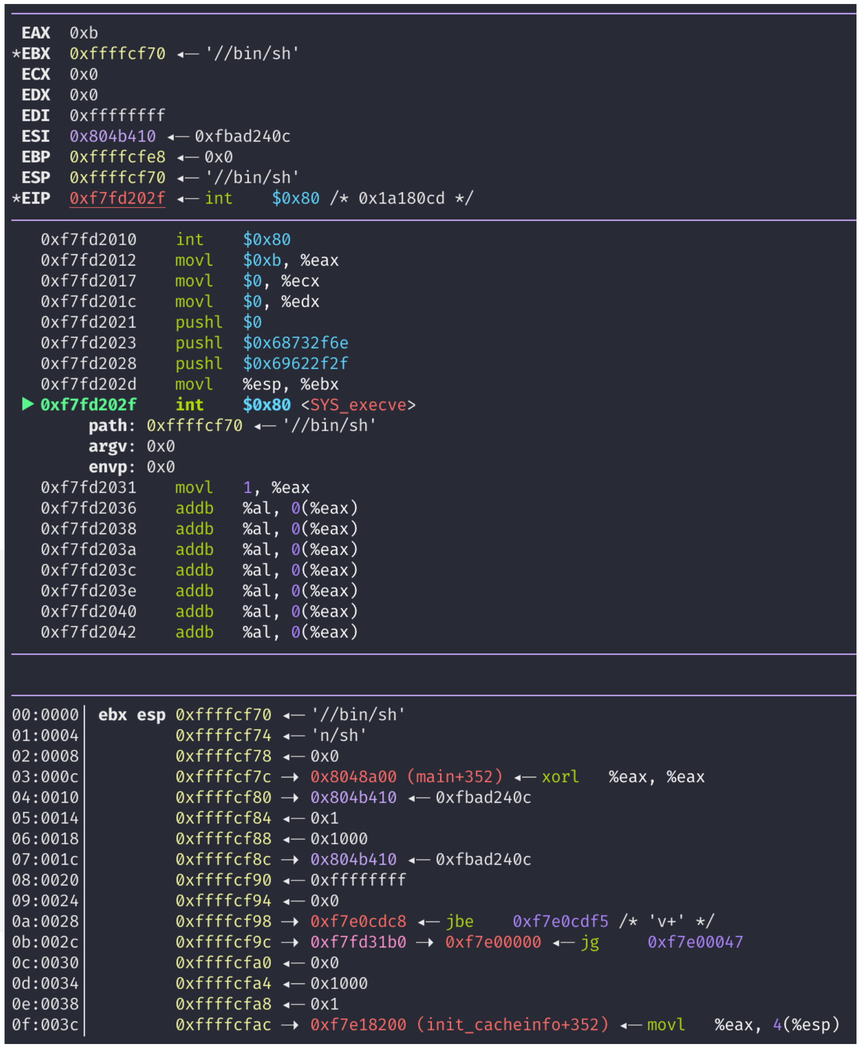
How to Create a String (‘/bin/sh’)?

`%ebx` = address of “`/bin/sh`”

- Use Stack

```
push $0                // why?
push $0x67832f6e       // “n/sh”
push $0x69622f2f       // “//bi”


 mov %esp, %ebx
```



```
 EAX  0xb
*EBX  0xffffcf70 ◂— '//bin/sh'
 ECX  0x0
 EDX  0x0
 EDI  0xffffffff
 ESI  0x804b410 ◂— 0xfbad240c
 EBP  0xffffcfe8 ◂— 0x0
 ESP  0xffffcf70 ◂— '//bin/sh'
*EIP  0xf7fd202f ◂— int    $0x80 /* 0x1a180cd */

   0xf7fd2010    int    $0x80
   0xf7fd2012    movl   $0xb, %eax
   0xf7fd2017    movl   $0, %ecx
   0xf7fd201c    movl   $0, %edx
   0xf7fd2021    pushl  $0
   0xf7fd2023    pushl  $0x68732f6e
   0xf7fd2028    pushl  $0x69622f2f
   0xf7fd202d    movl   %esp, %ebx
 ▶ 0xf7fd202f    int    $0x80 <SYS_execve>
        path: 0xffffcf70 ◂— '//bin/sh'
        argv: 0x0
        envp: 0x0
   0xf7fd2031    movl   1, %eax
   0xf7fd2036    addb   %al, 0(%eax)
   0xf7fd2038    addb   %al, 0(%eax)
   0xf7fd203a    addb   %al, 0(%eax)
   0xf7fd203c    addb   %al, 0(%eax)
   0xf7fd203e    addb   %al, 0(%eax)
   0xf7fd2040    addb   %al, 0(%eax)
   0xf7fd2042    addb   %al, 0(%eax)


00:0000│ ebx esp 0xffffcf70 ◂— '//bin/sh'
01:0004│         0xffffcf74 ◂— 'n/sh'
02:0008│         0xffffcf78 ◂— 0x0
03:000c│         0xffffcf7c —▸ 0x8048a00 (main+352) ◂— xorl   %eax, %eax
04:0010│         0xffffcf80 —▸ 0x804b410 ◂— 0xfbad240c
05:0014│         0xffffcf84 ◂— 0x1
06:0018│         0xffffcf88 ◂— 0x1000
07:001c│         0xffffcf8c —▸ 0x804b410 ◂— 0xfbad240c
08:0020│         0xffffcf90 ◂— 0xffffffff
09:0024│         0xffffcf94 ◂— 0x0
0a:0028│         0xffffcf98 —▸ 0xf7e0cdc8 ◂— jbe    0xf7e0cdf5 /* 'v+' */
0b:002c│         0xffffcf9c —▸ 0xf7fd31b0 —▸ 0xf7e00000 ◂— jg     0xf7e00047
0c:0030│         0xffffcfa0 ◂— 0x0
0d:0034│         0xffffcfa4 ◂— 0x1000
0e:0038│         0xffffcfa8 ◂— 0x1
0f:003c│         0xffffcfac —▸ 0xf7e18200 (init_cacheinfo+352) ◂— movl   %eax, 4(%esp)
```

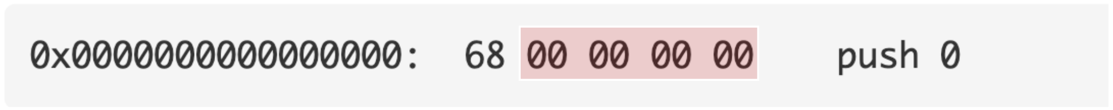
Controlling the Attack Surface

- Limit/monitor the input channels to the program (or service)
  - Command-line
  - STDIN, input files
  - Environment variables

- Limit/monitor the contents of the input
  - Are you string?
  - Are you too lengthy?
  - Do you contain any control characters? – e.g., “;”

Shellcode with Zero-bytes

- push $0



```
0x0000000000000000:  68 00 00 00 00    push 0
```

- Standard functions will cut off your shellcode

```
scanf(), strcpy(), fgets() …
```

Removing Zero from Your Shellcode

- You still need *Zeros* for your code
- Do do you create and use Zeros?

Many Tricks

```
mov $0x41414141, %eax
sub $0x41414141, %eax
```

```
xor %eax, %eax
mov %eax, %ebx
```

- Try it from shell-storm
  - **NOTE:** shell-storm assembler/disassembler only understands ‘Intel’ syntax 😔

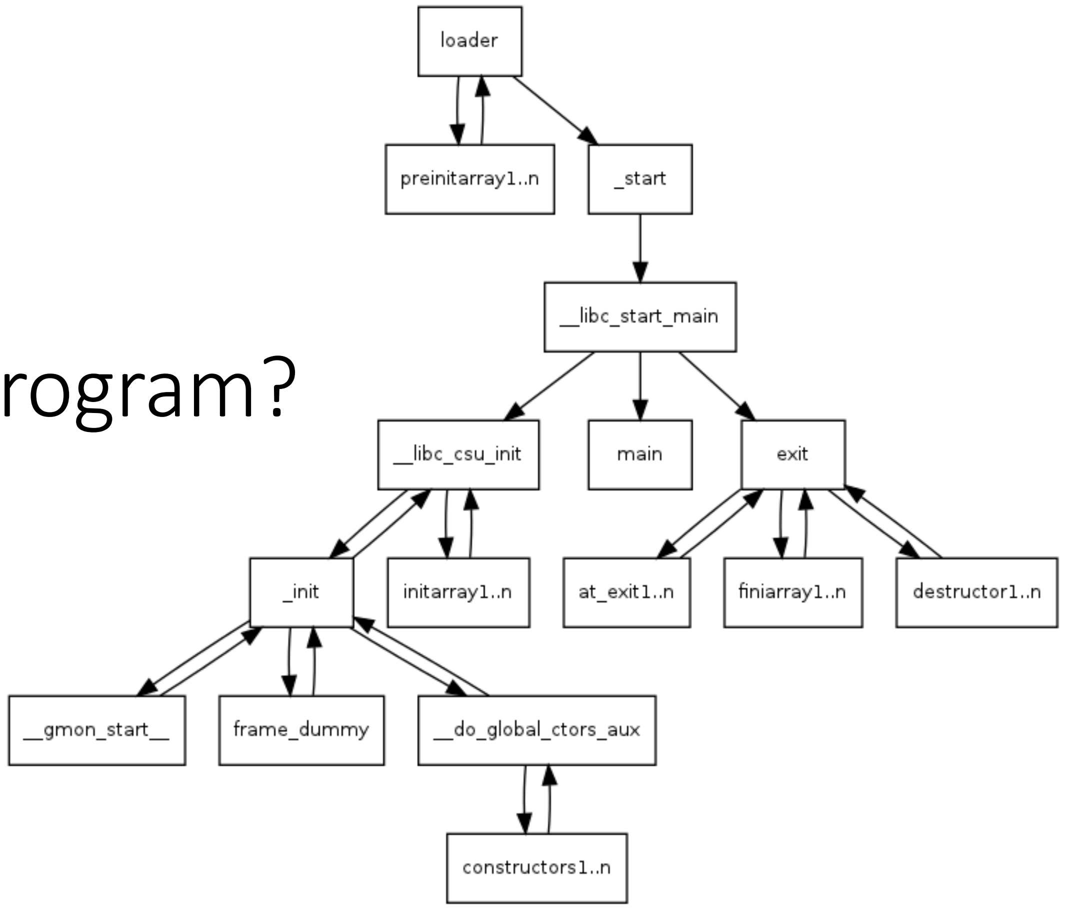
http://shell-storm.org/online/Online-Assembler-and-Disassembler/

Loading You Payload

- Leverages *program inputs*
  - Many sanitizations for their inputs
  - *E.g.,* how program cut string end?

- Program will only accept
  - ASCII characters
  - Alphanumeric characters
  - Limits in input length
  - No escape characters …

How Linux Runs a Program?



Stack for Communication

- Used for
  - Storing local variables
    - Your input buffer could be here…
  - Passing function arguments
  - Storing return address
  - Storing frame (base) pointer (i.e., saved %ebp)
    - Chaining stack frames
- What others?
  - filename, ARGV, ENVP, AUX

How Linux Runs a Program?

- Reading:
  - **How programs get run: ELF binaries**
  - https://lwn.net/Articles/631631/
  - ELF (Executable and Linkable Format)
- Basics:

```
execve(char *name, char** argv, char **envp)
execve(“program path”, argument_vectors, environment_pointers)
execve(“/bin/sh”, {“sh”, NULL}, {”SHELL=sh”, “TERM=xterm”, …, NULL});
```

How Linux Runs a Program?

- Reading:
  - **How programs get run: ELF binaries**
  - https://lwn.net/Articles/631631/
  - ELF (Executable and Linkable Format)
- Basics:

```
execve(char *name, char** argv, char **envp)
execve(“program path”, argument_vectors, environment_pointers)
execve(“/bin/sh”, {“sh”, NULL}, {”SHELL=sh”, “TERM=xterm”, …, NULL});
```

`execve(char *name, char** argv, char **envp)`

1. Program path: program to execute; could be an ELF file

   /bin/sh , /bin/ls, /usr/bin/python

2. Argument vectors: list of arguments, each as string (`char **argv`)
   - When running “python a.py 1 2 3”,

```
argv[0] = “python”
argv[1] = “a.py”
argv[2] = “1”
argv[3] = ”2”
argv[4] = “3”
```

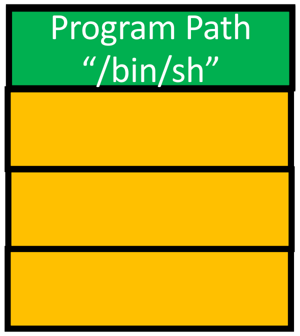
3. Environment Variables: list of environments variables, each as string (char **envp)

```
“TERM=xterm”
“SHELL=sh”
“EDITOR=/usr/bin/vim”
```

1. Load the Executable

- From the program path (1st arg to execve), the kernel loads code and data to the memory space

```
execve("/bin/sh", ….)
```



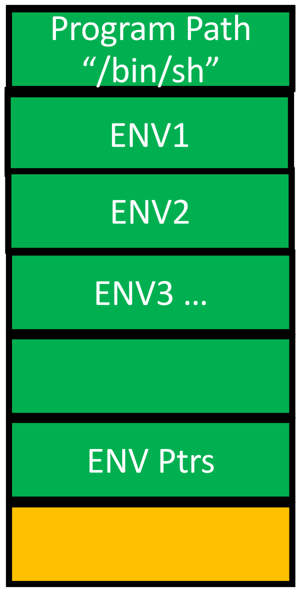
- Code section
  - Usually starts from 0x8048000 (32 bit), or 0x400000 (64 bit)

```
readelf -a /bin/sh
```

- **Will store the path name of the program at the bottom of the stack**

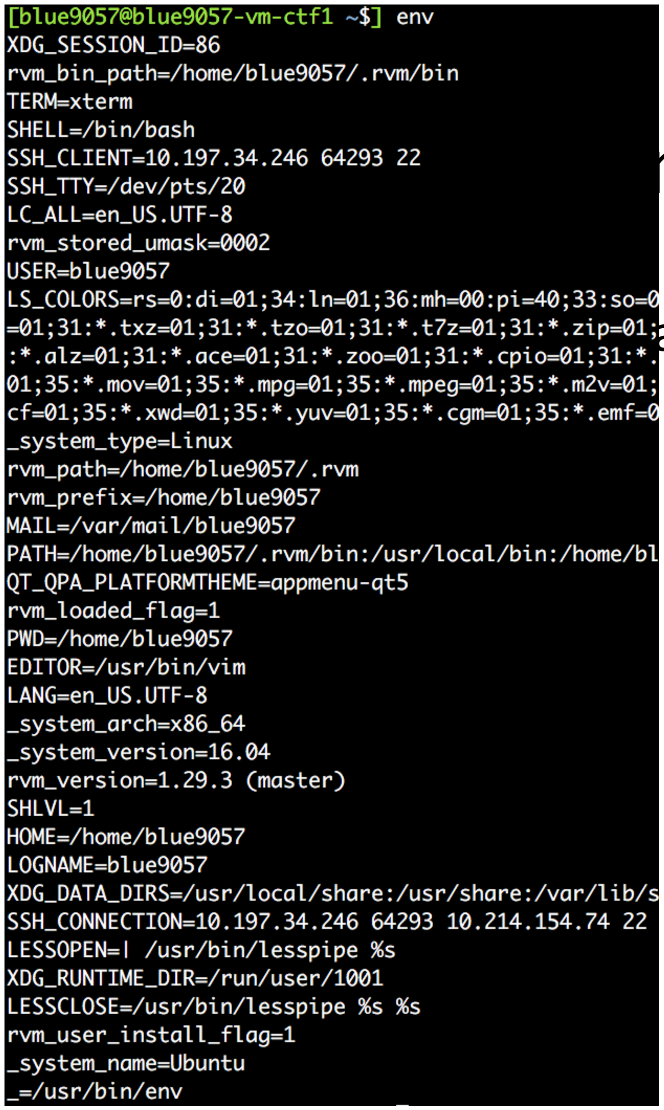
2. Set Environment Variables

- Set environmental variables to the current stack.

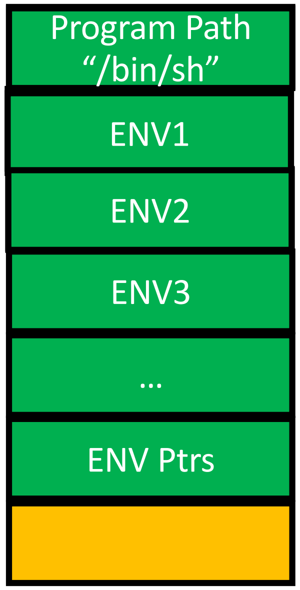


ment Variables

ariables to the current stack.



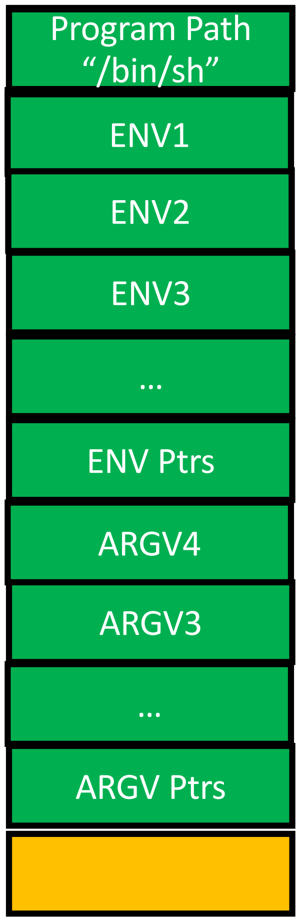
```
[blue9057@blue9057-vm-ctf1 ~$] env
XDG_SESSION_ID=86
rvm_bin_path=/home/blue9057/.rvm/bin
TERM=xterm
SHELL=/bin/bash
SSH_CLIENT=10.197.34.246 64293 22
SSH_TTY=/dev/pts/20
LC_ALL=en_US.UTF-8
rvm_stored_umask=0002
USER=blue9057
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=0
=01;31:*.txz=01;31:*.tzo=01;31:*.t7z=01;31:*.zip=01;
:*.alz=01;31:*.ace=01;31:*.zoo=01;31:*.cpio=01;31:*.
01;35:*.mov=01;35:*.mpg=01;35:*.mpeg=01;35:*.m2v=01;
cf=01;35:*.xwd=01;35:*.yuv=01;35:*.cgm=01;35:*.emf=0
_system_type=Linux
rvm_path=/home/blue9057/.rvm
rvm_prefix=/home/blue9057
MAIL=/var/mail/blue9057
PATH=/home/blue9057/.rvm/bin:/usr/local/bin:/home/bl
QT_QPA_PLATFORMTHEME=appmenu-qt5
rvm_loaded_flag=1
PWD=/home/blue9057
EDITOR=/usr/bin/vim
LANG=en_US.UTF-8
_system_arch=x86_64
_system_version=16.04
rvm_version=1.29.3 (master)
SHLVL=1
HOME=/home/blue9057
LOGNAME=blue9057
XDG_DATA_DIRS=/usr/local/share:/usr/share:/var/lib/s
SSH_CONNECTION=10.197.34.246 64293 10.214.154.74 22
LESSOPEN=| /usr/bin/lesspipe %s
XDG_RUNTIME_DIR=/run/user/1001
LESSCLOSE=/usr/bin/lesspipe %s %s
rvm_user_install_flag=1
_system_name=Ubuntu
_=/usr/bin/env
```



3. Set Argument Vectors

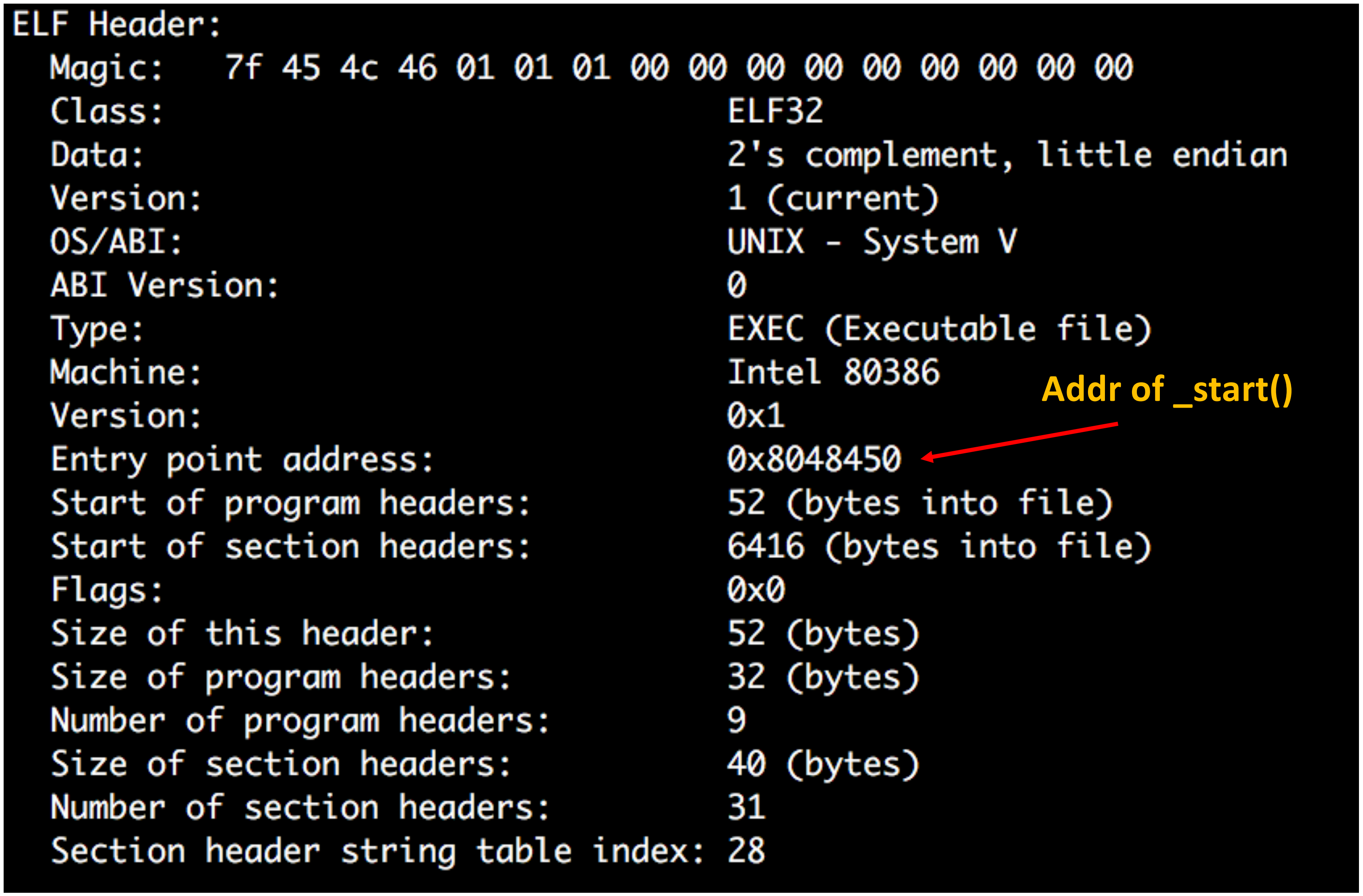
- Set argument vectors to the current stack.
  - When running “python a.py 1 2 3”,

  argv[0] = “python”
  argv[1] = “a.py”
  argv[2] = “1”
  argv[3] = ”2”
  argv[4] = “3”



4. Call _start

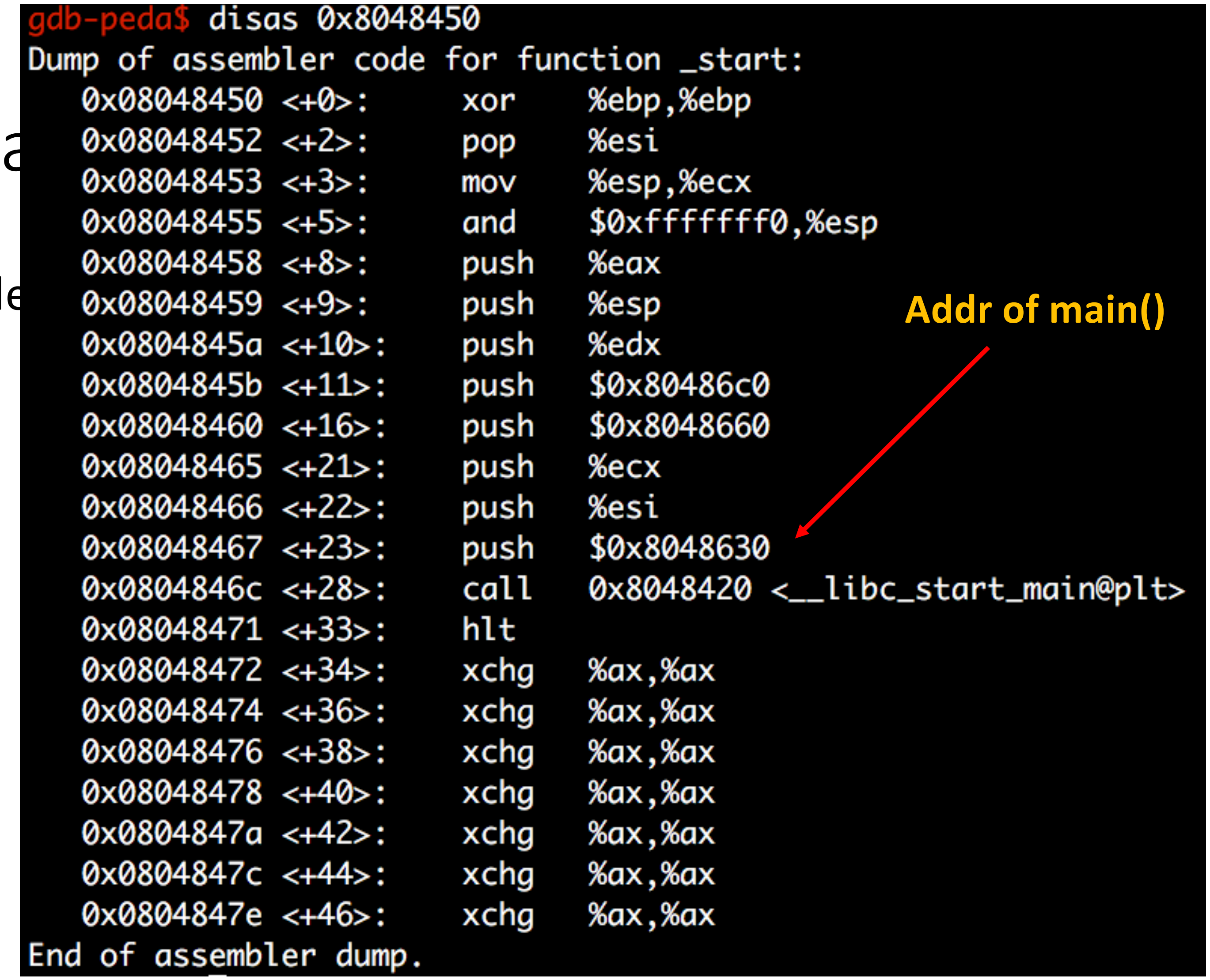
```
readelf -a …
```



```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x8048450
  Start of program headers:          52 (bytes into file)
  Start of section headers:          6416 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         9
  Size of section headers:           40 (bytes)
  Number of section headers:         31
  Section header string table index: 28
```

4. Ca

- reade



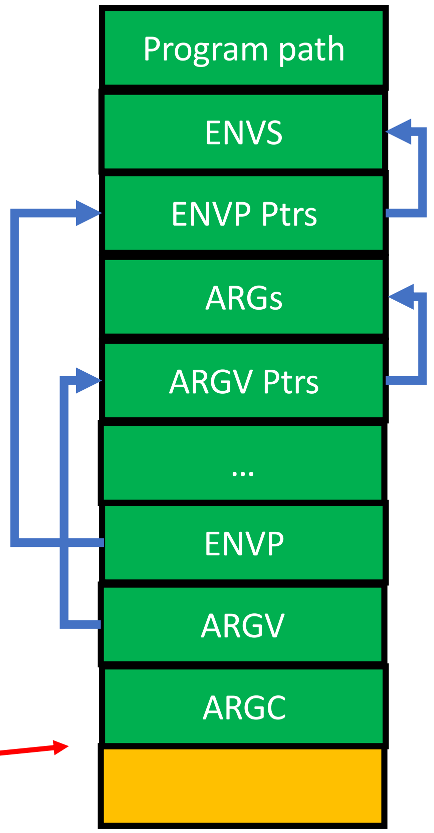
```
gdb-peda$ disas 0x8048450
Dump of assembler code for function _start:
   0x08048450 <+0>:     xor    %ebp,%ebp
   0x08048452 <+2>:     pop    %esi
   0x08048453 <+3>:     mov    %esp,%ecx
   0x08048455 <+5>:     and    $0xfffffff0,%esp
   0x08048458 <+8>:     push   %eax
   0x08048459 <+9>:     push   %esp
   0x0804845a <+10>:    push   %edx
   0x0804845b <+11>:    push   $0x80486c0
   0x08048460 <+16>:    push   $0x8048660
   0x08048465 <+21>:    push   %ecx
   0x08048466 <+22>:    push   %esi
   0x08048467 <+23>:    push   $0x8048630
   0x0804846c <+28>:    call   0x8048420 <__libc_start_main@plt>
   0x08048471 <+33>:    hlt
   0x08048472 <+34>:    xchg   %ax,%ax
   0x08048474 <+36>:    xchg   %ax,%ax
   0x08048476 <+38>:    xchg   %ax,%ax
   0x08048478 <+40>:    xchg   %ax,%ax
   0x0804847a <+42>:    xchg   %ax,%ax
   0x0804847c <+44>:    xchg   %ax,%ax
   0x0804847e <+46>:    xchg   %ax,%ax
End of assembler dump.
```

4. Call _start

- readelf -a …
- Calls `__libc_start_main`

- And then, `libc_start_main` sets the address of argv, envp, then calls `main(argc, argv, envp)`

The stack of main() starts at here:



Check with GDB

```
b main
r
x/100x $esp
```

Backup