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

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
More than 10,992 new Windows viruses and worms in the last half year of 2005.
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%).
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%).
853 “high severity” vulnerabilities disclosed in last half of 2005.
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.
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 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.)
Some common code exploitation techniques:

- Buffer Overflows
- Format String Vulnerabilities
- Race conditions
- Code injection (SQL)
- XSS scripting
Some common code exploitation techniques:

- Buffer Overflows
- Format String Vulnerabilities
- 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

```c
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

**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

```assembly
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 inAssembler

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
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
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
  addl $12,%esp
  xorl %eax,%eax
  leave
  ret
Function Calls in Assembler

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

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

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

Diagram of function call stack and registers:
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

```assembly
code:
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
```

**Example:**
- `pushl %ebp`: Push the base pointer onto the stack.
- `movl %esp,%ebp`: Move the stack pointer to the base pointer.
- `subl $4,%esp`: Subtract 4 from the stack pointer.
- `movl $1,-4(%ebp)`: Move 1 to the memory address `-4(%ebp)`.
- `call foo`: Call the function `foo`.
- `addl $12,%esp`: Add 12 to the stack pointer.
- `xorl %eax,%eax`: XOR the eax register with itself.
- `leave`: Leave the function.
- `ret`: Return from the function.
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
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

```assembly
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
```

 esp + ebp

 sfp

 rip

 sfp

 1

 3

 2

 1

 00 00 00 41

 42
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

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
Vulnerable Code: `foo.c`

```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;
}
```
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'`
- ./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'``
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'`

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

```
Matthias Vallentin
Beyond Stack Smashing
46 / 142
```
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 Ingredients

Injected code has generally two components:

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

Matthias Vallentin | Beyond Stack Smashing | 54 / 142
NOP sliding [Phr49-14]

```c
char shellcode[] =
    "\xeb\x1f"              /* jmp 0x1f     ...    */
    "\xe8\xdc\xff\xff\xff"  /* call -0x24           (5) */
    "/bin/sh";
```

Matthias Vallentin
Beyond Stack Smashing 55 / 142
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
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 \( \leq N \) instead of \( < N \) (and vice versa).
- giving something next to the person who should 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 $\leq N$ instead of $< N$ (and vice versa).
- giving something next to the person who should 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).
void foo()
{
    char buf[256];
    int i;

    for (i = 0; i <= 256; i++)
        buf[i] = 0xff;
}
Frame Pointer Overwrite

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

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

```assembly
leave
ret
```

```assembly
movl %ebp,%esp ; esp := modified SFP (mSFP)
popl %ebp
```

```assembly
ret:
popl %eip ; eip := mSFP + 4
```
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`.

```c
leave
ret

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

```assembly
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:   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`.

```c
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]
The Exploitation Technique [Phr55-8]

sfp = ebp-8

\[ \text{sfp} = \text{ebp} - 8 \]
The Exploitation Technique [Phr55-8]
The Exploitation Technique [Phr55-8]
The Exploitation Technique [Phr55-8]

1. Generation: Stack-based Overflows
2. Generation: Off-by-Ones and Frame Pointer Overwrites
3. Generation: BSS Overflows
4. Generation: Heap Overflows

```
rip
sfp
buf
main()
foo()
payload addr
shellcode
NOPs
0xbafffe40
0xbafffe44
0xbafffe48
0xbafffe4c
0xbafffe3c
0xbafffe38

eip = rip
```

```
leave: (foo)
    movl %ebp,%esp
    popl %ebp
    ret:
    popl %eip
```
The Exploitation Technique [Phr55-8]

1. Generation: Stack-based Overflows
2. Generation: Off-by-Ones and Frame Pointer Overwrites
3. Generation: BSS Overflows
4. Generation: Heap Overflows
The Exploitation Technique [Phr55-8]

1. Generation: Stack-based Overflows
2. Generation: Off-by-Ones and Frame Pointer Overwrites
3. Generation: BSS Overflows
4. Generation: Heap Overflows

The diagram shows the memory layout and exploit code. The exploit involves overwriting the return address to a shellcode section. The code snippet at the bottom shows the machine code instructions:

```assembly
leave: (main)
    movl %ebp, %esp
    popl %ebp
ret:
    popl %eip
```
The Exploitation Technique [Phr55-8]

Matthias Vallentin  Beyond Stack Smashing  78 / 142
1. Generation: Stack-based Overflows
2. Generation: Off-by-Ones and Frame Pointer Overwrites
3. Generation: BSS Overflows
4. Generation: Heap Overflows

The Exploitation Technique [Phr55-8]

```
main()
foo()
payload addr
shellcode
NOPs
leave: (main)
movl %ebp, %esp
popl %ebp
ret:
popl %eip
```
The Exploitation Technique [Phr55-8]

1. Generation: Stack-based Overflows

2. Generation: Off-by-Ones and Frame Pointer Overwrites

3. Generation: BSS Overflows

4. Generation: Heap Overflows

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

Matthias Vallentin

Beyond Stack Smashing

80 / 142
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
**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).

---

```
Matthias Vallentin
Beyond Stack Smashing
```

---

```
0xc0000000
high address

0x08048000
low address
```
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);
    ...
}

"/tmp/file"
BSS
Data
argv[1] Stack
buf
*tmpfile
buf:
int main(int argc, char *argv[]) {
    FILE *tmpfd;
    static char buf[24];
    static char *tmpfile;
    tmpfile =="/tmp/file";
    gets(buf);
    fputs(buf, tmpfd);
    ...
}

buf:

ruth::0:0::/:/bin/sh #
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
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** 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 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 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).

---

![Free Chunk Diagram]

- **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**: forward pointer.
- **bk**: backward pointer.
- **data**: data from previous chunk.

---

![Allocated Chunk Diagram]

- **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**: forward pointer.
- **bk**: backward pointer.
- **data**: data from previous chunk.
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).

---

**Free Chunk**

- **chunk**
- **mem**
- **nextchunk**

<table>
<thead>
<tr>
<th>prev_size</th>
<th>size</th>
<th>PREV_INUSE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>&gt;= 0</td>
</tr>
</tbody>
</table>

**Allocated Chunk**

- **chunk**
- **mem**
- **nextchunk**

<table>
<thead>
<tr>
<th>prev_size</th>
<th>size</th>
<th>PREV_INUSE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>&gt;= 0</td>
</tr>
</tbody>
</table>

- **data**
- **data from previous chunk**
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 similar size are grouped into **bins**.
- **fd/bk** pointers to navigate through double links.
Chunks in Memory

- allocated chunk
- allocated chunk
- free chunk
- allocated chunk
- free chunk
- allocated chunk
- wilderness chunk
Removing Chunks from a Bin: `unlink()`

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

Removing Chunks from a Bin: \texttt{unlink()}
Removing Chunks from a Bin: `unlink()`

```c
#define unlink(P, BK, FD)
{
    BK = P->bk;
    FD = P->fd;
    FD->bk = BK;
    BK->fd = FD;
}
```
Removing Chunks from a Bin: `unlink()`

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

FD + 12 = BK
```
#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

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

Similar to the stack, we can overwrite internal management information.

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

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

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.

   → Evaluation of the `PREV_INUSE` bit of the third chunk.
Vulnerable Code

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

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

### Diagram

```
+----+----+----+----+----+----+----+----+----+----+
|size|    | size|    | size|    | PREV_INUSE|
+----+----+----+----+----+----+----+----+----+----+
|    | fd  |    | 32 | 31 |    | PREV_INUSE|
|    | bk  |    |    |    |    | PREV_INUSE|
+----+----+----+----+----+----+----+----+----+----+
|    | prev_size|    | 34 | 33 | 32 | 31 |
|    | bk      |    |    |    |    |    |
|    | prev_size|    | 38 | 37 | 36 | 35 |
|    | size    |    |    |    |    |    |
|    | fd      |    | 32 | 31 | 30 | 39 |
|    | bk      |    |    |    |    |    |
|    | size    |    |    |    |    |    |
|    | fd      |    |    |    |    |    |
|    | bk      |    |    |    |    |    |
|    | prev_size|    |    |    |    |    |
|    | size    |    |    |    |    |    |
|    | fd      |    |    |    |    |    |
|    | bk      |    |    |    |    |    |
```

---

Matthias Vallentin
Beyond Stack Smashing
Vulnerable Code

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

1. `free(buf1)` looks at `PREV_INUSE` of chunk #3.
unlink() Vulnerability (cont’d)

1. free(buf1) looks at \textit{PREV\_INUSE} of chunk #3.
2. unlink() on chunk #2.

Vulnerable Code

\begin{itemize}
  \item free(buf1) looks at \\textit{PREV\_INUSE} of chunk #3.
  \item unlink() on chunk #2.
\end{itemize}
Vulnerable Code

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

```
1. unlink() looks at PREV_INUSE of chunk #3.
2. unlink() on chunk #2.
3. P->fd->bk = P->bk
```
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`
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`
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.
  ⇒ 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);
...

```
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. $FD = \&\text{free}() - 12$

```
FD = &free() - 12
BK = buf2 + 8
FD->bk = BK
BK->fd = FD
```

- `FD` points to the free list.
- `BK` points to the buffer.
- Overwriting `FD` with `BK` modifies the free list.
- Overwriting `BK` with `FD` modifies the buffer.

```
prev_size
size
PREV_INUSE
fd
bk
prev_size
size
PREV_INUSE
&free() - 12
buf2 + 8
shellcode
42 42 42 42
ff ff ff \0
```

- `prev_size` and `size` point to the free list.
- `PREV_INUSE` is set to indicate free status.
- `shellcode` is on high addresses.
- `buf1`, `buf2`, and `buf3` are used to manipulate the memory.

Matthias Vallentin
Beyond Stack Smashing
Practical Exploitation

Vulnerable Code

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

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

```
buf1
prev_size
size
PREV_INUSE
fd
bk
prev_size
size
PREV_INUSE
&free() - 12
buf2 + 8
shellcode
42 42 42 42
ff ff ff 0
fd
bk
buf3
```
Practical Exploitation

Vulnerable Code

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

```
FD = &free() - 12
BK = buf2 + 8
FD->bk = BK
```

```
prev_size
size     | PREV_INUSE
---------|-----------
fd       |           
bk       |           
prev_size
size     | PREV_INUSE
---------|-----------
&free() - 12
buf2 + 8
shellcode
42    | 42    | 42    | 42
ff    | ff    | ff    | \0
fd
bk
buf1
buf2
buf3
```
Vulnerable Code

1. \( FD = \&\text{free()} - 12 \)
2. \( BK = \text{buf2} + 8 \)
3. \( FD\rightarrow bk = BK \)
   \( \Rightarrow \&\text{free()} \) is now \&shellcode

Practical Exploitation
Vulnerable Code

1. \( FD = &\text{free()} - 12 \)
2. \( BK = \text{buf2} + 8 \)
3. \( FD->bk = BK \)
   \( \rightarrow \&\text{free()} \text{ is now } \&\text{shellcode} \)
4. \( BK->fd = FD \)

Diagram:

- \( \text{prev_size} \)
- \( \text{size} \)
- \( \text{PREV_INUSE} \)
- \( \text{fd} \)
- \( \text{bk} \)
- \( \text{buf1} \)
- \( \text{buf2} \)
- \( \text{buf3} \)
- \( \text{&free()} - 12 \)
- \( \text{buf2 + 8} \)
- \( \text{shellcode} \)
- \( 42 \)
- \( ff \)
- \( \0 \)
### Practical Exploitation

#### Vulnerable Code

1. \( FD = \&\text{free()} - 12 \)
2. \( BK = \text{buf2} + 8 \)
3. \( FD->bk = BK \rightarrow \&\text{free()} \) is now \&shellcode
4. \( BK->fd = FD \rightarrow \) 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$
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 engineering: 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
Buffer overflows are just the beginning.
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
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.

- **John R. Levine.**
Linkers & Loaders.

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