This time

We will begin our 1st section: **Software Security**

By investigating **Buffer overflows** and other memory safety vulnerabilities

- History
- Memory layouts
- Buffer overflow fundamentals
Software security

• Security is a form of dependability
  • Does the code do “what it should”
  • To this end, we follow the software lifecycle

• Distinguishing factor: an *active, malicious* attacker

• Attack model
  • The developer is trusted
  • But the attacker can provide any inputs
    - Malformed strings
    - Malformed packets
    - etc.

What harm could an attacker possibly cause?
screensaver --prompt="Don't unlock plz"

Don't unlock plz

Locked by dml

press ctrl-c to logout
screensaver --prompt="Don’t unlock pretty plz"

Don't unlock pretty plz
Locked by dml
press ctrl-c to logout
screensaver --prompt="Don’t unlock plz
Locked by dml
press ctrl-c to logout
screensaver -prompt="Under maintenance;\Do not interrupt
Locked by dml
press ctrl-c to logout"
We’re going to focus on C

C is still very popular

http://www.tiobe.com
We’re going to focus on C

Many mission critical systems are written in C

• Most kernels & OS utilities
  • fingerd
  • X windows server

• Many high-performance servers
  • Microsoft IIS
  • Microsoft SQL server

• Many embedded systems
  • Mars rover

• But the techniques apply more broadly
  • Wiibrew: “Twilight Hack” exploits buffer overflow when saving the name of Link’s horse, Epona
We’re going to focus on C

The harm can be substantial

• Morris worm
  • Propagated across machines (too aggressively, thanks to a bug)
  • One way it propagated was a buffer [overflow] attack against a vulnerable version of [fingerd] on VAXes
    • Sent a special string to the finger daemon, which caused it to execute code that created a new worm copy
    • Didn’t check OS: caused Suns running BSD to crash
  • End result: $10-100M in damages, probation, community service
We’re going to focus on C

The harm can be substantial


• Morris worm
  • Propagated across machines (too aggressively, thanks to a bug)
  • One way it propagated was a **buffer overflow attack** against a vulnerable version of `fingerd` on VAXes
    • Sent a special string to the finger daemon, which caused it to execute code that created a new worm copy
    • Didn’t check OS: caused Suns running BSD to crash
  • End result: $10-100M in damages, probation, community service

Robert Morris is now a professor at MIT
We’re going to focus on C

The harm can be substantial

• CodeRed
  • Exploited an overflow in the MS-IIS server
  • 300,000 machines infected in 14 hours
We’re going to focus on C

The harm can be substantial

- CodeRed
  - Exploited an overflow in the MS-IIS server
  - 300,000 machines infected in 14 hours
We’re going to focus on C

The harm can be substantial

- SQL Slammer
  - Exploited an overflow in the MS-SQL server
  - 75,000 machines infected in 10 minutes
23-Year-Old X11 Server Security Vulnerability Discovered

Posted by Unknown Lamer on Wednesday January 08, 2014 @10:11AM
from the stack-smashing-for-fun-and-profit dept.

An anonymous reader writes

"The recent report of X11/X.Org security in bad shape rings more truth today. The X.Org Foundation announced today that they've found a X11 security issue that dates back to 1991. The issue is a possible stack buffer overflow that could lead to privilege escalation to root and affects all versions of the X Server back to X11R5. After the vulnerability being in the code-base for 23 years, it was finally uncovered via the automated cppcheck static analysis utility."

There's a scanf used when loading BDF fonts that can overflow using a carefully crafted font. Watch out for those obsolete early-90s bitmap fonts.
An anonymous reader writes

"The recent report of X11/X.Org security in bad shape rings more truth today. The X.Org Foundation announced today that they've found a X11 security issue that dates back to 1991. The issue is a possible stack buffer overflow that could lead to privilege escalation to root and affects all versions of the X Server back to X11R5. After the vulnerability being in the code-base for 23 years, it was finally uncovered via the automated cppcheck static analysis utility."

There's a scanf used when loading BDF fonts that can overflow using a carefully crafted font. Watch out for those obsolete early-90s bitmap fonts.
GHOST: glibc vulnerability introduced in 2000, only just announced two days ago
Buffer overflows are prevalent

Percent of *all* vulnerabilities

Buffer overflows are prevalent

Total number of buffer overflow vulnerabilities

This class
This class E-voting

Brief Listing of the Top 25

This is a brief listing of the Top 25 items, using the general ranking.

NOTE: 16 other weaknesses were considered for inclusion in the Top 25, but their general scores were not high enough. They are listed in a separate "On the Cusp" page.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Score</th>
<th>ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>93.8</td>
<td>CWE-89</td>
<td>Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')</td>
</tr>
<tr>
<td>[2]</td>
<td>83.3</td>
<td>CWE-78</td>
<td>Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')</td>
</tr>
<tr>
<td>[3]</td>
<td>79.0</td>
<td>CWE-120</td>
<td>Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')</td>
</tr>
<tr>
<td>[4]</td>
<td>77.7</td>
<td>CWE-79</td>
<td>Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')</td>
</tr>
<tr>
<td>[6]</td>
<td>76.8</td>
<td>CWE-862</td>
<td>Missing Authorization</td>
</tr>
<tr>
<td>[7]</td>
<td>75.0</td>
<td>CWE-798</td>
<td>Use of Hard-coded Credentials</td>
</tr>
<tr>
<td>[8]</td>
<td>75.0</td>
<td>CWE-311</td>
<td>Missing Encryption of Sensitive Data</td>
</tr>
<tr>
<td>[9]</td>
<td>74.0</td>
<td>CWE-434</td>
<td>Unrestricted Upload of File with Dangerous Type</td>
</tr>
<tr>
<td>[10]</td>
<td>73.8</td>
<td>CWE-807</td>
<td>Reliance on Untrusted Inputs in a Security Decision</td>
</tr>
<tr>
<td>[11]</td>
<td>73.1</td>
<td>CWE-250</td>
<td>Execution with Unnecessary Privileges</td>
</tr>
<tr>
<td>[12]</td>
<td>70.1</td>
<td>CWE-352</td>
<td>Cross-Site Request Forgery (CSRF)</td>
</tr>
</tbody>
</table>
This is a brief listing of the Top 25 items, using the general ranking.

NOTE: 16 other weaknesses were considered for inclusion in the Top 25, but their general scores were not high enough. They are listed in a separate "On the Cusp" page.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Score</th>
<th>ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>93.8</td>
<td>CWE-89</td>
<td>Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')</td>
</tr>
<tr>
<td>[2]</td>
<td>83.3</td>
<td>CWE-78</td>
<td>Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')</td>
</tr>
<tr>
<td>[3]</td>
<td>79.0</td>
<td>CWE-120</td>
<td>Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')</td>
</tr>
<tr>
<td>[4]</td>
<td>77.7</td>
<td>CWE-79</td>
<td>Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')</td>
</tr>
<tr>
<td>[6]</td>
<td>76.8</td>
<td>CWE-862</td>
<td>Missing Authorization</td>
</tr>
<tr>
<td>[7]</td>
<td>75.0</td>
<td>CWE-798</td>
<td>Use of Hard-coded Credentials</td>
</tr>
<tr>
<td>[8]</td>
<td>75.0</td>
<td>CWE-311</td>
<td>Missing Encryption of Sensitive Data</td>
</tr>
<tr>
<td>[9]</td>
<td>74.0</td>
<td>CWE-434</td>
<td>Unrestricted Upload of File with Dangerous Type</td>
</tr>
<tr>
<td>[10]</td>
<td>73.8</td>
<td>CWE-807</td>
<td>Reliance on Untrusted Inputs in a Security Decision</td>
</tr>
<tr>
<td>[11]</td>
<td>73.1</td>
<td>CWE-250</td>
<td>Execution with Unnecessary Privileges</td>
</tr>
<tr>
<td>[12]</td>
<td>70.1</td>
<td>CWE-352</td>
<td>Cross-Site Request Forgery (CSRF)</td>
</tr>
</tbody>
</table>
Our goals

- Understand how these attacks work, and how to defend against them

- These require knowledge about:
  - The compiler
  - The OS
  - The architecture

Analyzing security requires a whole-systems view
Memory layout
Refresher

• How is program data laid out in memory?

• What does the stack look like?

• What effect does calling (and returning from) a function have on memory?

• We are focusing on the Linux process model
  • Similar to other operating systems
All programs are stored in memory
All programs are stored in memory
All programs are stored in memory

The process’s view of memory is that it owns all of it
All programs are stored in memory

The *process’s view* of memory is that it owns all of it.

In reality, these are *virtual addresses*; the OS/CPU map them to physical addresses.
The instructions themselves are in memory
The instructions themselves are in memory

```
0x4b7 mov %esp,%ebp
0x4b9 push %ebp
0x4c1 push %ecx
0x4c2 sub $0x224,%esp
0x4c5 ...
```

```
0x4c2 sub $0x224,%esp
0x4c1 push %ecx
0x4bf mov %esp,%ebp
0x4be push %ebp
0x4bf ...
```
Data’s location depends on how it’s created
Data’s location depends on how it’s created

```
static const int y = 10;
```
Data’s location depends on how it’s created

- **Uninit’d data**: `static int x;`
- **Init’d data**: `static const int y=10;`
- **Text**:
Data’s location depends on how it’s created

Known at compile time

- Uninit’d data
- Init’d data
- Text

static int x;
static const int y=10;
Data’s location depends on how it’s created

```
static int x;
static const int y=10;
```

4G

- cmdline & env
- Uninit’d data
- Init’d data
- Text

Known at compile time

$0xffffffff$

$0x00000000$
Data’s location depends on how it’s created

Set when process starts

Known at compile time

cmdline & env

Uninit’d data

Init’d data

Text

static int x;
static const int y=10;

4G

0xffffffff

0x00000000

0xffffffff

0x00000000
Data’s location depends on how it’s created.

Set when process starts:
- cmdline & env
- Stack
- Uninit’d data
- Init’d data
- Text

Known at compile time:
- static int x;
- static const int y=10;

Code example:
```c
int f() {
    int x;
    ...
}
```

Memory layout:
- 0xffffffff
- 0x00000000
- Stack
- Uninit’d data
- Init’d data
- Text
Data’s location depends on how it’s created

- **Set when process starts**
  - 4G
  - cmdline & env
  - Stack
  - Heap
  - Uninit’d data
  - Init’d data
  - Text

- **Known at compile time**

  ```
  #include <stdio.h>
  int f() {
      int x;
      ...
  
  malloc(sizeof(long));
  static int x;
  static const int y=10;
  ```

  ```
  0xffffffff
  int f() {
      int x;
      ...
  
  malloc(sizeof(long));
  static int x;
  static const int y=10;
  ```

  ```
  0x00000000
  ```
Data’s location depends on how it’s created

Set when process starts

Runtime

Known at compile time

---

4G

0xffffffff

0x00000000

4G

cmdline & env

Stack

Heap

Uninit’d data

Init’d data

Text

int f() {
    int x;
    ...
    malloc(sizeof(long));
    static int x;
    static const int y=10;
}

0xffffffff

0x00000000
We are going to focus on runtime attacks

Stack and heap grow in opposite directions
We are going to focus on runtime attacks

Stack and heap grow in opposite directions

Compiler provides instructions that adjusts the size of the stack at runtime
We are going to focus on runtime attacks

**Stack and heap grow in opposite directions**

Compiler provides instructions that adjusts the size of the stack at runtime
We are going to focus on runtime attacks

**Stack and heap grow in opposite directions**

Compiler provides instructions that adjusts the size of the stack at runtime

0x000000000000 - 0xffffffff

```
Stack pointer
```

```
push 1
push 2
push 3
```
We are going to focus on runtime attacks.

**Stack and heap grow in opposite directions**

Compiler provides instructions that adjusts the size of the stack at runtime.
We are going to focus on runtime attacks

**Stack and heap grow in opposite directions**

Compiler provides instructions that adjusts the size of the stack at runtime.

```
push 1
push 2
push 3
```
We are going to focus on runtime attacks

Stack and heap grow in opposite directions

Compiler provides instructions that adjusts the size of the stack at runtime

```
push 1
push 2
push 3
```
We are going to focus on runtime attacks

Stack and heap grow in opposite directions

Compiler provides instructions that adjusts the size of the stack at runtime

Stack pointer

push 1
push 2
push 3
We are going to focus on runtime attacks.

Stack and heap grow in opposite directions.

Compiler provides instructions that adjusts the size of the stack at runtime.

```
push 1
push 2
push 3
```
We are going to focus on runtime attacks

Stack and heap grow in opposite directions

Compiler provides instructions that adjusts the size of the stack at runtime

Stack pointer

push 1
push 2
push 3
We are going to focus on runtime attacks.

Stack and heap grow in opposite directions.

Compiler provides instructions that adjusts the size of the stack at runtime.

```
push 1
push 2
push 3
```
We are going to focus on runtime attacks

**Stack and heap grow in opposite directions**

Compiler provides instructions that adjusts the size of the stack at runtime

```
push 1
push 2
push 3
return
```
We are going to focus on runtime attacks.

**Stack and heap grow in opposite directions**

Compiler provides instructions that adjusts the size of the stack at runtime.

```
push 1
push 2
push 3
return
```
We are going to focus on runtime attacks

**Stack and heap grow in opposite directions**

Compiler provides instructions that adjusts the size of the stack at runtime

```
push 1
push 2
push 3
return
```

Heap: apportioned by the OS; managed in-process by `malloc`

Stack: pointer

0x000000000 0xffffffff
We are going to focus on runtime attacks

Stack and heap grow in opposite directions

Compiler provides instructions that adjusts the size of the stack at runtime

apportioned by the OS; managed in-process by `malloc`

Focusing on the stack for now
Stack layout when calling functions

• What do we do when we call a function?
  • What data need to be stored?
  • Where do they go?

• How do we return from a function?
  • What data need to be restored?
  • Where do they come from?

Code examples
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int loc2;
    int loc3;
}

0x000000000000    0xffffffff

caller’s data
Stack layout when calling functions

```c
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int loc2;
    int loc3;
}
```

Arguments pushed in reverse order of code
Stack layout when calling functions

```c
#include <stdio.h>

void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int loc2;
    int loc3;
}
```

Local variables pushed in the same order as they appear in the code,
Arguments pushed in reverse order of code.
Stack layout when calling functions

```c
void func(char *arg1, int arg2, int arg3) {
    char loc1[4]
    int loc2;
    int loc3;
}
```

Local variables pushed in the same order as they appear in the code

Arguments pushed in reverse order of code
void func(char *arg1, int arg2, int arg3) {
    char loc1[4]
    int loc2;
    int loc3;
    ...
    loc2++;
    ...
}
Accessing variables

```c
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int  loc2;
    int  loc3;
    ...
    loc2++;  // Q: Where is (this) loc2?
    ...
}
```

0x000000000000  0xffffffffffffffff

<table>
<thead>
<tr>
<th>...</th>
<th>loc2</th>
<th>loc1</th>
<th>???</th>
<th>???</th>
<th>arg1</th>
<th>arg2</th>
<th>arg3</th>
<th>caller's data</th>
</tr>
</thead>
</table>
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int loc2;
    int loc3;
    ...
    loc2++;  Q: Where is (this) loc2?
    ...
}
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int loc2;
    int loc3;
    ...
    loc2++;
    ...
}

Q: Where is (this) loc2?

Undecidable at compile time
void func(char *arg1, int arg2, int arg3) {
    char loc1[4]
    int loc2;
    int loc3;
    ...
    loc2++;  \textbf{Q: Where is (this) loc2?}
    ...
}

- I don't know where loc2 is,
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int loc2;
    int loc3;
    ...
    loc2++;
    ...
}

Q: Where is (this) loc2?

Undecidable at compile time
- I don’t know where loc2 is,
- and I don’t know how many args
void func(char *arg1, int arg2, int arg3) {
    char loc1[4]
    int loc2;
    int loc3;
    ...
    loc2++;
    ...
}

Q: Where is (this) loc2?

Undecidable at compile time

- I don’t know where loc2 is,
- and I don’t know how many args
- *but* loc2 is always 8B before “???”'s
Accessing variables

```c
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int loc2;
    int loc3;
    ...
    loc2++;  // Q: Where is (this) loc2?
    ...
}
```

- I don’t know where loc2 is,
- and I don’t know how many args
- *but* loc2 is *always* 8B before “???”'s
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int loc2;
    int loc3;
    ...
    loc2++;
    ...
}

Q: Where is (this) loc2?
- I don’t know where loc2 is,
- and I don’t know how many args
- but loc2 is always 8B before “???”s
void func(char *arg1, int arg2, int arg3) {
    char loc1[4]
    int loc2;
    int loc3;
    ...
    loc2++;  Q: Where is (this) loc2?
    ...
}

Stack frame for this call to func

- I don’t know where loc2 is,
- and I don’t know how many args
- but loc2 is always 8B before “???”s
void func(char *arg1, int arg2, int arg3) {
    char loc1[4]
    int loc2;
    int loc3;
    ...
    loc2++;  
    ...
}

Q: Where is (this) loc2?
A: -8(%ebp)

Frame pointer

- I don’t know where loc2 is,
- and I don’t know how many args
- but loc2 is always 8B before “???”s
Notation

%ebp  A memory address

(%ebp)  The value at memory address %ebp
(like dereferencing a pointer)
Notation

%ebp  A memory address

(%ebp)  The value at memory address %ebp
       (like dereferencing a pointer)
Notation

0xbfff03b8  %ebp  A memory address

(%ebp)  The value at memory address %ebp
(like dereferencing a pointer)
Notation

0xbfff03b8  \%ebp  A memory address

(\%ebp)  The value at memory address \%ebp
(like dereferencing a pointer)
Notation

0xbfff03b8 %ebp A memory address

0xbfff0720 (%ebp) The value at memory address %ebp (like dereferencing a pointer)
Notation

0xbfff03b8  %ebp  A memory address

0xbfff0720  (%ebp)  The value at memory address %ebp (like dereferencing a pointer)

pushl %ebp
Notation

0xbfff03b8  %ebp  A memory address

0xbfff0720  (%ebp)  The value at memory address %ebp (like dereferencing a pointer)

pushl %ebp
**Notation**

0xbfff03b8  \%^ebp\%^  A memory address

0xbfff0720 (\%^ebp\%^)  The value at memory address \%^ebp\%^ (like dereferencing a pointer)

```
pushl %ebp
```
Notation

0xbfffo3b8 \%ebp A memory address

0xbfffo720 (\%ebp) The value at memory address \%ebp (like dereferencing a pointer)

pushl \%ebp
Notation

0xbfff03b8 %ebp A memory address

0xbfff0720 (%ebp) The value at memory address %ebp (like dereferencing a pointer)

pushl %ebp
Notation

0xbfff03b8 %ebp A memory address

0xbfff0720 (%ebp) The value at memory address %ebp (like dereferencing a pointer)

```assembly
pushl %ebp
movl %esp %ebp /* %ebp = %esp */
```
Notation

0xbfff03b8 %ebp  A memory address

0xbfff0720 (%ebp) The value at memory address %ebp (like dereferencing a pointer)

```assembly
pushl %ebp
movl %esp %ebp /* %ebp = %esp */
```
Notation

%ebp  A memory address

(%ebp) The value at memory address %ebp (like dereferencing a pointer)

pushl %ebp
movl %esp %ebp  /* %ebp = %esp */
Notation

- \%ebp: A memory address
- \( \%ebp \): The value at memory address \%ebp (like dereferencing a pointer)

```
pushl \%ebp
movl \%esp \%ebp /* \%ebp = \%esp */
```
**Notation**

- `%ebp` A memory address
- `( %ebp )` The value at memory address `%ebp` (like dereferencing a pointer)

**Code Snippet**

```
pushl %ebp
movl %esp %ebp  /* %ebp = %esp */
```

**Memory Addresses**

- `0xbfff03b8`
- `0xbfff0200`
- `0xbfff0720`
- `0x00000000`
- `0xffffffff`

**Diagram**

- `%ebp` is pushed onto the stack, and `%ebp` is updated to point to `%esp`.
Returning from functions

int main()
{
    ...
    func("Hey", 10, -3);
    ...
}

Stack frame for this call to func
Returning from functions

```c
int main()
{
    ...
    func("Hey", 10, -3);
    ...
}
```

Stack frame for *this* call to `func`
Returning from functions

```c
int main()
{
    ...
    func("Hey", 10, -3);
    ...
}
```

Stack frame for this call to `func`
Returning from functions

```c
int main()
{
    ...
    func("Hey", 10, -3);
    ...
}
```

**Q: How do we restore `%ebp`?**

**Stack frame for this call to `func`**

%ebp
Returning from functions

```c
int main()
{
    ...
    func("Hey", 10, -3);
    ...
    Q: How do we restore %ebp?
}
```

Q: How do we restore %ebp?
Return from functions

```c
int main()
{
    ...
    func("Hey", 10, -3);
    ...
    Q: How do we restore %ebp?
}
```

Q: How do we restore %ebp?

Stack frame for this call to func
Returning from functions

int main()
{
    ...
    func("Hey", 10, -3);
    ...
    Q: How do we restore %ebp?
}

1. Push %ebp before locals

Stack frame for this call to func
Returning from functions

```
int main()
{
    ...
    func("Hey", 10, -3);
    ...
    Q: How do we restore %ebp?
}
```

1. Push %ebp before locals
2. Set %ebp to current %esp
Returning from functions

```c
int main()
{
    ...
    func("Hey", 10, -3);
    ...
    Q: How do we restore %ebp?
}
```

Stack frame for this call to func

1. Push %ebp before locals
2. Set %ebp to current %esp
3. Set %ebp to (%ebp) at return
int main()
{
    ...
    func(“Hey”, 10, -3);
    ...
}
Returning from functions

```c
int main()
{
    ...
    func("Hey", 10, -3);
    ...  Q: How do we resume here?
}
```

Q: How do we resume here?

Stack frame for this call to `func`

%ebp for this call to `func`
The instructions themselves are in memory

```
0xffffffff
```

```
0x4a7 movl $0x0,%eax
0x4a2 call <func>
0x49b movl $0x804..,%esp
0x493 movl $0xa,0x4(%esp)
  ... 
```

```
0x00000000
```
The instructions themselves are in memory.

```
... 0x4a7 mov $0x0,%eax
0x4a2 call <func>
0x49b movl $0x804..,(%esp)
0x493 movl $0xa,0x4(%esp)
... 0x0
```
The instructions themselves are in memory

```
... 0x4a7 movl $0x0,%eax
0x4a2 call <func>
0x49b movl $0x804..,%esp
0x493 movl $0xa,0x4(%esp)
...%eip
```
The instructions themselves are in memory

Text

4G

0xffffffff

0x00000000

0xffffffff

%eip

... 0x4a7 mov $0x0,%eax
0x4a2 call <func>
0x49b movl $0x804..,%esp
0x493 movl $0xa,0x4(%esp)
...
The instructions themselves are in memory

4G

0x000000000

Text

0xffffffff

...  
0x5bf mov %esp,%ebp  
0x5be push %ebp  
...  

%eip

...  
0x4a7 mov $0x0,%eax  
0x4a2 call <func>  
0x49b movl $0x804..,(%esp)  
0x493 movl $0xa,0x4(%esp)  
...
The instructions themselves are in memory

\[
0xffffffff
\]

\[
... 
0x5bf \text{mov} \%esp,\%ebp
0x5be \text{push} \%ebp
...
\]

\[
... 
0x4a7 \text{mov} \%eax
0x4a2 \text{call} <\text{func}>
0x49b \text{movl} \%eax,0x804..,(\%esp)
0x493 \text{movl} \%eax,0x4(%esp)
...
\]
The instructions themselves are in memory

```
0xffffffff
...
0x5bf mov %esp,%ebp
0x5be push %ebp
...
```

```
0x4a7 mov $0x0,%eax
0x4a2 call <func>
0x49b movl $0x804..,(%esp)
0x493 movl $0xa,0x4(%esp)
...
```
The instructions themselves are in memory

4G

0x0000000000

0xffffffff

0x5bf mov %esp,%ebp
0x5be push %ebp
...

0x4a7 mov $0x0,%eax
0x4a2 call <func>
0x49b movl $0x804..,(%esp)
0x493 movl $0xa,0x4(%esp)
...
Returning from functions

```c
int main()
{
    ...
    func("Hey", 10, -3);
    ...
    Q: How do we resume here?
}
```

Stack frame for this call to `func`
Returning from functions

```c
int main()
{
    ...
    func("Hey", 10, -3);
    ...
    Q: How do we resume here?
}
```

Stack frame for this call to `func`

%ebp for this call to `func`

Push next %eip before call
Returning from functions

```c
int main()
{
    ...
    func("Hey", 10, -3);
    ...
    Q: How do we resume here?
}
```

Stack frame for this call to `func`

Push next `%eip` before call
Returning from functions

```c
int main()
{
    ...
    func("Hey", 10, -3);
    ...
    Q: How do we resume here?
}
```

Q: How do we resume here?

Set `%eip` to `4(%ebp)` at return

Push next `%eip` before call

Stack frame for this call to `func`

caller's data

%ebp
%ebp
%ebp

loc1 loc2 %ebp %eip arg1 arg2 arg3 caller's data

0x0000000000

0xffffffff

0xffffffff
Stack and functions: Summary
Stack and functions: Summary

**Calling function:**

1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: `%eip+something`
3. **Jump to the function’s address**
Stack and functions: Summary

Calling function:
1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: %eip+something
3. **Jump to the function’s address**

Called function:
4. **Push the old frame pointer** onto the stack: %ebp
5. **Set frame pointer** %ebp to where the end of the stack is right now: %esp
6. **Push local variables** onto the stack; access them as offsets from %ebp
Stack and functions: Summary

Calling function:
1. Push arguments onto the stack (in reverse)
2. Push the return address, i.e., the address of the instruction you want run after control returns to you: %eip+something
3. Jump to the function’s address

Called function:
4. Push the old frame pointer onto the stack: %ebp
5. Set frame pointer %ebp to where the end of the stack is right now: %esp
6. Push local variables onto the stack; access them as offsets from %ebp

Returning function:
7. Reset the previous stack frame: %ebp = (%ebp)
8. Jump back to return address: %eip = 4(%ebp)
Buffer overflows
Buffer overflows from 10,000 ft

• **Buffer** =
  • Contiguous set of a given data type
  • Common in C
    - All strings are buffers of char’s

• **Overflow** =
  • Put more into the buffer than it can hold

• Where does the extra data go?

• Well now that you’re experts in memory layouts…
A buffer overflow example

```c
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}

int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
```
A buffer overflow example

```c
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}

int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
```
A buffer overflow example

```c
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}

int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
```
A buffer overflow example

```c
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}

int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
```
A buffer overflow example

```c
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}

int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
```
A buffer overflow example

```c
void func(char *arg1) {
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}

int main() {
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
```
A buffer overflow example

```c
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}

int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
```
A buffer overflow example

```c
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}

int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
```

```
Me! \0

Auth 4d 65 21 00 %eip &arg1
```

buffer
A buffer overflow example

```c
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}

int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
```

Upon return, sets `%ebp` to 0x0021654d

```
M e ! \0
```

<table>
<thead>
<tr>
<th>Auth</th>
<th>4d 65 21 00</th>
<th><code>%eip</code></th>
<th>&amp;arg1</th>
</tr>
</thead>
<tbody>
<tr>
<td>buffer</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A buffer overflow example

```c
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}

int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
```

Upon return, sets %ebp to 0x0021654d

```
Me ! \0
```

<table>
<thead>
<tr>
<th>Auth</th>
<th>4d 65 21 00</th>
<th>%eip</th>
<th>&amp;arg1</th>
</tr>
</thead>
<tbody>
<tr>
<td>buffer</td>
<td>SEGFAULT (0x00216551)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A buffer overflow example

```c
void func(char *arg1)
{
    int authenticated = 0;
    char buffer[4];
    strcpy(buffer, arg1);
    if(authenticated) { ... }
}

int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ... 
}
```
A buffer overflow example

```c
void func(char *arg1)
{
    int authenticated = 0;
    char buffer[4];
    strcpy(buffer, arg1);
    if(authenticated) { ... }
}

int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ... 
}
```
A buffer overflow example

```c
void func(char *arg1)
{
    int authenticated = 0;
    char buffer[4];
    strcpy(buffer, arg1);
    if(authenticated) { ... }
}

int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
```
A buffer overflow example

```c
void func(char *arg1)
{
    int authenticated = 0;
    char buffer[4];
    strcpy(buffer, arg1);
    if(authenticated) { ... }
}

int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ... 
}
```
A buffer overflow example

```c
void func(char *arg1)
{
    int authenticated = 0;
    char buffer[4];
    strcpy(buffer, arg1);
    if(authenticated) { ... 
}

int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
```
A buffer overflow example

```c
void func(char *arg1)
{
    int authenticated = 0;
    char buffer[4];
    strcpy(buffer, arg1);
    if(authenticated) { ... }
}

int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ... 
}
```

Authenticated

```
00 00 00 00 %ebp %eip &arg1
```
A buffer overflow example

```c
void func(char *arg1)
{
    int authenticated = 0;
    char buffer[4];
    strcpy(buffer, arg1);
    if(authenticated) { ... }
}

int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ... 
}
```
A buffer overflow example

```c
void func(char *arg1)
{
    int authenticated = 0;
    char buffer[4];
    strcpy(buffer, arg1);
    if(authenticated) { ...
}

int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
```

```
A  u  t  h
00 00 00 00 %ebp %eip &arg1
```

buffer authenticated
A buffer overflow example

```c
void func(char *arg1)
{
    int authenticated = 0;
    char buffer[4];
    strcpy(buffer, arg1);
    if(authenticated) { ...
}

int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
```

<table>
<thead>
<tr>
<th>Auth</th>
<th>4d 65 21 00</th>
<th>%ebp</th>
<th>%eip</th>
<th>&amp;arg1</th>
</tr>
</thead>
<tbody>
<tr>
<td>buffer</td>
<td>authenticated</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A buffer overflow example

```c
void func(char *arg1)
{
    int authenticated = 0;
    char buffer[4];
    strcpy(buffer, arg1);
    if(authenticated) { ... }
}

int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ... 
}
```

Code still runs; user now ‘authenticated’

```
Me! \0
```

```
M  e  !  \0
```

```
Auth  4d 65 21 00  ebp  eip  &arg1
```

buffer  authenticated
```c
void vulnerable()
{
    char buf[80];
    gets(buf);
}
```
void vulnerable()
{
    char buf[80];
    gets(buf);
}

void still_vulnerable()
{
    char *buf = malloc(80);
    gets(buf);
}
void safe()
{
    char buf[80];
    fgets(buf, 64, stdin);
}
void safe()
{
    char buf[80];
    fgets(buf, 64, stdin);
}

void safer()
{
    char buf[80];
    fgets(buf, sizeof(buf), stdin);
}
IE's Role in the Google-China War

By Richard Adhikari
TechNewsWorld
01/15/10 12:25 PM PT

The hack attack on Google that set off the company's ongoing standoff with China appears to have come through a zero-day flaw in Microsoft's Internet Explorer browser. Microsoft has released a security advisory, and researchers are hard at work studying the exploit. The attack appears to consist of several files, each a different piece of malware.

Computer security companies are scurrying to cope with the fallout from the Internet Explorer (IE) flaw that led to cyberattacks on Google and its corporate and individual customers.

The zero-day attack that exploited IE is part of a lethal cocktail of malware that is keeping researchers very busy.

"We're discovering things on an up-to-the-minute basis, and we've seen about a dozen files dropped on infected PCs so far," Dmitri Alperovitch, vice president of research at McAfee Labs, told TechNewsWorld.

The attacks on Google, which appeared to originate in China, have sparked a feud between the Internet giant and the nation's government over censorship, and it could result in Google pulling away from its business dealings in the country.

Pointing to the Flaw

The vulnerability in IE is an invalid pointer reference, Microsoft said in security advisory 979352, which it issued on Thursday. Under certain conditions, the invalid pointer can be accessed after an object is deleted, the advisory states. In specially crafted attacks, like the ones launched against Google and its customers, IE can allow remote execution of code when the flaw is exploited.
User-supplied strings

• In these examples, we were providing our own strings

• But they come from users in myriad ways
  • Text input
  • Packets
  • Environment variables
  • File input…
What’s the worst that could happen?

```c
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}
```

00 00 00 00  ebp  eip  &myst

buffer
What’s the worst that could happen?

```c
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}
```

`strcpy` will let you write as much as you want (til a `\0`)
What’s the worst that could happen?

```c
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}
```

`strcpy` will let you write as much as you want (til a `\0`)
What’s the worst that could happen?

```c
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}
```

strcpy will let you write as much as you want (til a ‘\0’)

All ours!

What could you write to memory to wreak havoc?
Code injection
High-level idea

```c
void func(char *arg1)
{
    char buffer[4];
    sprintf(buffer, arg1);
    ...
}
```

... 00 00 00 00  ebp  eip  &arg1  ...

buffer
High-level idea

```c
void func(char *arg1)
{
    char buffer[4];
    sprintf(buffer, arg1);
    ...
}
```

(1) Load my own code into memory
High-level idea

```c
void func(char *arg1)
{
    char buffer[4];
    sprintf(buffer, arg1);
    ...
}
```

1. Load my own code into memory
2. Somehow get `%eip` to point to it
High-level idea

```c
void func(char *arg1)
{
    char buffer[4];
    sprintf(buffer, arg1);
    ...
}
```

(1) Load my own code into memory
(2) Somehow get `%eip` to point to it
void func(char *arg1)
{
    char buffer[4];
    sprintf(buffer, arg1);
    ...
}

(1) Load my own code into memory
(2) Somehow get %eip to point to it
This is nontrivial

• Pulling off this attack requires getting a few things really right (and some things sorta right)

• Think about what is tricky about the attack
  • The key to defending it will be to make the hard parts really hard
Challenge 1

Loading code into memory

• It must be the machine code instructions (i.e., already compiled and ready to run)

• We have to be careful in how we construct it:
  • It can’t contain any all-zero bytes
    - Otherwise, sprintf / gets / scanf / … will stop copying
    - How could you write assembly to never contain a full zero byte?
  • It can’t make use of the loader (we’re injecting)
  • It can’t use the stack (we’re going to smash it)
What kind of code would we want to run?

- **Goal:** full-purpose shell
  - The code to launch a shell is called “shell code”
  - It is nontrivial to do it in a way that works as injected code
    - No zeroes, can’t use the stack, no loader dependence
- **Goal:** privilege escalation
  - Ideally, they go from guest (or non-user) to root
Shellcode

#include <stdio.h>
int main( ) {
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
Shellcode

```c
#include <stdio.h>
int main( ) {
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

Assembly

```
xorl %eax, %eax
pushl %eax
pushl $0x68732f2f
pushl $0x6e69622f
movl %esp,%ebx
pushl %eax
...```

Shellcode

```c
#include <stdio.h>
int main( ) {
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

Assembly

```assembly
xorl %eax, %eax
pushl %eax
pushl $0x68732f2f
pushl $0x6e69622f
movl %esp,%ebx
movl %esp,%eax
...```
Shellcode

```c
#include <stdio.h>
int main( ) {
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

```
xorl %eax, %eax
pushl %eax
pushl $0x68732f2f
pushl $0x6e69622f
movl %esp,%ebx
pushl %eax
...
```

```
"\x31\xc0"
"\x50"
"\x68""/\sh"
"\x68""/bin"
"\x89\xe3"
"\x50"
...
```
Shellcode

```c
#include <stdio.h>
int main( ) {
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

Assembly

```
xorl %eax, %eax
pushl %eax
pushl $0x68732f2f
pushl $0x6e69622f
movl %esp,%ebx
pushl %eax
...```

Machine code

```
\x31\xc0
\x50
\x68"//sh"
\x68"/bin"
\x89\xe3
\x50
...
```
Privilege escalation

• Permissions later, but for now…

• Recall that each file has:
  • Permissions: read / write / execute
  • For each of: owner / group / everyone else

• Consider a service like passwd
  • Owned by root (and needs to do root-y things)
  • But you want any user to be able to run it
Effective userid

• Userid = the user who ran the process

• Effective userid = what is used to determine what access the process has

• Consider passwd:
  • getuid() will return you (real userid)
  • seteuid(0) to set the effective userid to root
    - It’s allowed to because root is the owner

• What is the potential attack?
Effective userid

• Userid = the user who ran the process

• Effective userid = what is used to determine what access the process has

• Consider passwd:
  • getuid() will return you (real userid)
  • seteuid(0) to set the effective userid to root
    - It’s allowed to because root is the owner

• What is the potential attack?

If you can get a root-owned process to run setuid(0)/seteuid(0), then you get root permissions
Challenge 2

Getting our injected code to run

- We can't insert a "jump into my code" instruction
- We have to use whatever code is already running
Challenge 2

Getting our injected code to run

- We can’t insert a “jump into my code” instruction
- We have to use whatever code is already running

Thoughts?
Challenge 2

Getting our injected code to run

- We can’t insert a “jump into my code” instruction
- We have to use whatever code is already running

Thoughts?
Challenge 2

Getting our injected code to run

• We can’t insert a “jump into my code” instruction
• We have to use whatever code is already running

Thoughts?
Stack and functions: Summary

Calling function:
1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: %eip+something
3. **Jump to the function’s address**

Called function:
4. **Push the old frame pointer** onto the stack: %ebp
5. **Set frame pointer** %ebp to where the end of the stack is right now: %esp
6. **Push local variables** onto the stack; access them as offsets from %ebp

Returning function:
7. **Reset the previous stack frame**: %ebp = (%ebp)
8. **Jump back to return address**: %eip = 4(%ebp)
Hijacking the saved `%eip`
Hijacking the saved `%eip`

%eip  %ebp

Text  ...  00 00 00 00 %ebp  0xbff &arg1  ...  \x0f \x3c \x2f ...

buffer

0xbff
Hijacking the saved %eip
Hijacking the saved %eip

But how do we know the address?
Hijacking the saved `%eip`

What if we are wrong?

![Diagram showing memory layout with `%eip` and `%ebp` pointers, a buffer, and an offset `0xbff`.][1]

[1]: https://example.com/diagram.png
Hijacking the saved %eip

What if we are wrong?
Hijacking the saved %eip

What if we are wrong?
Hijacking the saved `%eip`

What if we are wrong?

This is most likely data, so the CPU will panic (Invalid Instruction)
Challenge 3

Finding the return address
Challenge 3

Finding the return address

• If we don’t have access to the code, we don’t know how far the buffer is from the saved ebp
Challenge 3

Finding the return address

• If we don’t have access to the code, we don’t know how far the buffer is from the saved `%ebp`

• One approach: just try a lot of different values!
Challenge 3

Finding the return address

• If we don’t have access to the code, we don’t know how far the buffer is from the saved \%ebp

• One approach: just try a lot of different values!

• Worst case scenario: it’s a 32 (or 64) bit memory space, which means $2^{32}$ ($2^{64}$) possible answers
Challenge 3

Finding the return address

• If we don’t have access to the code, we don’t know how far the buffer is from the saved %ebp

• One approach: just try a lot of different values!

• Worst case scenario: it’s a 32 (or 64) bit memory space, which means $2^{32} (2^{64})$ possible answers

• But without address randomization:
  • The stack always starts from the same, fixed address
  • The stack will grow, but usually it doesn’t grow very deeply (unless the code is heavily recursive)
gdb tutorial
Improving our chances: **nop sleds**

**nop** is a single-byte instruction (just moves to the next instruction)
Improving our chances: **nop** sleds

**nop** is a single-byte instruction (just moves to the next instruction)
Improving our chances: \texttt{nop} sleds

\texttt{nop} is a single-byte instruction (just moves to the next instruction)

Jumping \textit{anywhere} here will work
Improving our chances: \texttt{nop} sleds

\texttt{nop} is a single-byte instruction (just moves to the next instruction)

Jumping \textit{anywhere} here will work
Improving our chances: **nop sleds**

**nop** is a single-byte instruction (just moves to the next instruction)

Jumping *anywhere* here will work

Now we improve our chances of guessing by a factor of \#nops
Putting it all together

%eip padding good guess

Text ... 0xbdf nop nop nop ... \x0f \x3c \x2f ...

buffer nop sled malicious code
But it has to be *something*; we have to start writing wherever the input to `gets/etc.` begins.

Putting it all together

%eip

buffer

good guess

padding

nop sled malicious code

0xbdf nop nop nop ...

\x0f \x3c \x2f ...
Putting it all together

But it has to be *something*; we have to start writing wherever the input to `gets/etc.` begins.

```
0xbdf  nop  nop  nop ...
\0f  \3c  \2f ...
```

But it has to be something; we have to start writing wherever the input to `gets/etc.` begins.

```
buffer
```

```
Text ...
```

```
padding
good guess
%eip
```

```
nop sled malicious code
```
Putting it all together

But it has to be *something*; we have to start writing wherever the input to `gets/etc.` begins.

```
0xbff ... 0xbdf
```

```
nop nop nop ...
\x0f \x3c \x2f ...
```

```
Text ... padding good guess %eip
buffer

nop sled malicious code
```
Next time

Continuing with Software Security

More attacks, and Defenses

Required reading:
“StackGuard: Simple Stack Smash Protection for GCC”