Memory Corruption

The (almost) Complete History...

haroon meer - 2010
@haroonmeer | haroon@thinkst.com
Who?

- haroon meer
- thinkst?
- some papers, some books, some talks
- academic wannabe
Why?
Walking down memory lane, reading old exploits from '99 -- can someone write a history of code exec '95-2009?
Why?

The ROP discussion is amusing in the sense that our folklore gets republished, and then we are asked "what papers have you published"? :)

6:28 PM Feb 28th via web
Retweeted by 2 people

halvarflake
Each time your exploit heap spray, someone somewhere think he just pioneer an exploitation technique invented in the 90's

about 20 hours ago via web
Retweeted by 4 people

nicowaisman
Nico Waisman
Why?

twitter made me do it!
Why?

- de-mystify some of the otherwise mystical
- convince you that Solar Designer was skynet
Why?
(Some silly Stats)

Stack: 140
Heap: 74
Stack spraying is definitely impressive!
Caveats - Limits
Caveats - Limits

IA-32 series
Caveats - Myopia
Caveats - Myopia
Caveats - Compression Ratio

3328880 : 1
Disclosure, Bugs and Counts

VS.

thinkst applied research
Disclosed Bugs
Our Approach

Clearly naive initially
http://ilm.thinkst.com/folklore/
http://ilm.thinkst.com/folklore/
10/20/1995 - "How to Write Buffer Overflows"
Although only a private / internal release, Mudge (mudge@l0pht.com) published his document titled "How to write Buffer Overflows".[8]

Written primarily as a set of notes to himself, the document covers both the basics of the overflow and a rough introduction to writing shellcode. The document included an exploit for the syslogd [7] bug made public earlier.

12/3/1995 - splitv exploit published
DaveG and VicM of Avalon Research published an advisory (and exploit) for splitv on Linux 2-3.x. The vulnerability was due to an unbounded sprint call, which was exploited by an over long HOME environment variable.

11/8/1996 - Smashing the Stack Published
Aleph1 published what would become the most referenced paper on memory corruption attacks in Phrack49.[8]

From his introduction: ‘smash the stack’ [C programming] n. On many C implementations it is possible to corrupt the execution stack by writing past the end of an array declared auto in a routine. Code that does this is said to smash the stack, and can cause return from the routine to jump to a random address. This can produce some of the most insidious data-dependent bugs known to mankind.

3/21/1997 - Superprobe Exploit Published
Solar Designer overwrites function pointers to hijack execution flow.

4/22/1997 - DNS Poisoning QID Prediction
CORE and SNI report possible overflows due to bind ignoring MAXHOSTNAMELEN

1/20/97 - Stack Smashing Defenses Discussed
Bugtraq hosts a discussion on defenses against stack smashing.

3/21/1997 - Bypassing the non-exec Stack (ret-2-libc) - 8/10/1997
Solar Designer published the first known return-to-libc attack to overcome his own non-executable stack patch [9]. He demonstrated the technique using the ipr exploit against a Linux system with his non executable stack patch. His patch (for his patch) ensured that shared libraries are mmaped into regions containing a null byte to retard their use with unsafe string functions.
So at the end of this...

- You won't be able to suddenly use `free()` to obtain a 4-byte write anywhere primitive.
- You will understand what that means.
- You will be able to see:
  - When that was first used;
  - What prevents it’s use/abuse today;
Where did it start?
Fig. 1 Process Memory Regions
Memory Basics
Memory Basics

4 gig

0x00000000

User

Kernel

PageTable
Memory Basics

- User
- Kernel
- 0x00000000
- PageTable
- 4 gig

[Diagram of memory basics with a chip and a page table]
Multiple Processes

3 gig
Multiple Processes

3 gig

Kernel

User

User

User

0x00000000

0x00000000

0x00000000
Segments

4 gig

0x00000000

User

Kernel

Page Table

thinkst applied research
Segments

4 gig

0x00000000

User

Kernel
Segments

4 gig

User

Kernel

0x00000000

0x00000000
Segments

0x00000000
Text

0x00000000

User

Kernel

4 gig
Segments

- User
- Kernel
- 0x00000000
- 0x00000000
  - Text
  - Data

4 gig
Segments

4 gig

User

Kernel

0x00000000

Text

Data

...
Segments

User

Kernel

0x00000000

Grows Upwards

Text

Data

...
Segments

- Text
- Data
- Heap
- mmap (Shared Memory)

0x00000000 - grows upwards

User

Kernel

4 gig
Segments

- User
- Kernel
- Stack
- Heap
- mmap (Shared Memory)
- ...
- Data
- Text

0x00000000

Grows Upwards

Grows Downwards

4 gig
So what is code?
So what is code?
So what is code?
So what is code?
<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0040681D</td>
<td>C3 RETN</td>
</tr>
<tr>
<td>0040681E</td>
<td>55 PUSH EBP</td>
</tr>
<tr>
<td>0040681F</td>
<td>57 PUSH EDI</td>
</tr>
<tr>
<td>00406820</td>
<td>68 61742069 PUSH 69207461</td>
</tr>
<tr>
<td>00406825</td>
<td>73 20 JNB SHORT 00406847</td>
</tr>
<tr>
<td>00406827</td>
<td>74 68 JE SHORT 00406891</td>
</tr>
<tr>
<td>00406829</td>
<td>6973 F8 0083 IMUL ESI,DWORD PTR DS:[EBX-8],-39A7D00</td>
</tr>
<tr>
<td>00406830</td>
<td>0053 57 ADD BYTE PTR DS:[EBX+57],DL</td>
</tr>
<tr>
<td>00406833</td>
<td>BF 4EE640BB MOV EDI,8B40E64E</td>
</tr>
<tr>
<td>00406838</td>
<td>3BC7 CMP EAX,EDI</td>
</tr>
<tr>
<td>0040683A</td>
<td>BB 0000FFFF MOV EBX,FFFFF0000</td>
</tr>
<tr>
<td>0040683F</td>
<td>74 0D JE SHORT 0040684E</td>
</tr>
<tr>
<td>00406841</td>
<td>85C3 TEST EBX,EAX</td>
</tr>
<tr>
<td>00406843</td>
<td>74 09 JE SHORT 0040684E</td>
</tr>
<tr>
<td>00406845</td>
<td>F7D0 NOT EAX</td>
</tr>
<tr>
<td>00406847</td>
<td>C7 DC154100 MOV DWORD PTR DS:[4115DC],EAX</td>
</tr>
<tr>
<td>0040684C</td>
<td>EB 60 JMP SHORT 004068AE</td>
</tr>
<tr>
<td>0040684E</td>
<td>56 PUSH ESI</td>
</tr>
<tr>
<td>0040684F</td>
<td>8D45 F8 LEA EAX,[EBP-8]</td>
</tr>
<tr>
<td>00406852</td>
<td>5A PUSH FAX</td>
</tr>
</tbody>
</table>
Is this code?
Is this code?
Is this code?
User

Kernel

4 gig

0x00000000

0x00000000

Text

Data

Heap

... mmap (Shared Memory)

Stack

Grows Upwards

Grows Downwards
int function_1(int a, int b)
{
    int j;
    // do stuff
    return j;
}

int main(int argc, char **argv, char **envp)
{
    int i;
    i = function_1(1,2);
    printf("answer is %d", i)
    return i;
}
int function_1(int a, int b)
{
    int j;
    // do stuff
    return j;
}

int main(int argc, char **argv, char **envp)
{
    int i;
    i = function_1(1,2);
    printf("answer is %d", i)
    return i;
}
```c
int function_1(int a, int b)
{
    int j;
    // do stuff
    return j;
}

int main(int argc, char **argv, char **envp)
{
    int i;
    i = function_1(1,2);
    printf("answer is %d", i)
    return i;
}
```
int function_1(int a, int b)
{
    int j;
    // do stuff
    return j;
}

int main(int argc, char **argv, char **envp)
{
    int i;
    i = function_1(1,2);
    printf("answer is %d", i)
    return i;
}
```c
int function_1(int a, int b)
{
    int j;
    // do stuff
    return j;
}

int main(int argc, char **argv, char **envp)
{
    int i;
    i = function_1(1,2);
    printf("answer is %d", i);
    return i;
}
```
int function_1(int a, int b)
{
    int j;
    // do stuff
    return j;
}

int main(int argc, char **argv, char **envp)
{
    int i;
    i = function_1(1,2);
    printf("answer is %d", i);
    return i;
}
```c
int function_1(int a, int b)
{
    int j;
    // do stuff
    return j;
}

int main(int argc, char **argv, char **envp)
{
    int i;
    i = function_1(1, 2);
    printf("answer is \%d", i)
    return i;
}
```
int function_1(int a, int b)
{
    int j;
    // do stuff
    return j;
}

int main(int argc, char **argv, char **envp)
{
    int i;
    i = function_1(1,2);
    printf("answer is %d", i)
    return i;
}
int function_1(int a, int b)
{
    int j;
    // do stuff
    return j;
}

int main(int argc, char **argv, char **envp)
{
    int i;
    i = function_1(1,2);
    printf("answer is %d", i)
    return i;
}
int function_1(int a, int b) {
    int j;
    // do stuff
    return j;
}

int main(int argc, char **argv, char **envp) {
    int i;
    i = function_1(1,2);
    printf("answer is \%d", i)
    return i;
}
```c
int function_1(int a, int b)
{
    int j;
    // do stuff
    return j;
}

int main(int argc, char **argv, char **envp)
{
    int i;
    i = function_1(1,2);
    printf("answer is %d", i)
    return i;
}
```
int function_1(int a, int b)
{
    int j;
    // do stuff
    return j;
}

int main(int argc, char **argv, char **envp)
{
    int i;
    i = function_1(1,2);
    printf("answer is %d", i)
    return i;
}
int function_1(int a, int b)
{
    int j;
    // do stuff
    return j;
}

int main(int argc, char **argv, char **envp)
{
    int i;
    i = function_1(1, 2);
    printf("answer is %d", i)
    return i;
}
```c
int function_1(int a, int b) {
    int j;
    // do stuff
    return j;
}

int main(int argc, char **argv, char **envp) {
    int i;
    i = function_1(1,2);
    printf("answer is \%d", i)
    return i;
}
```
Classic Overflow

Where to go

Overflow Direction

Stack Grows Downwards
non-terminated strings

strcpy(buf1, buf2);
non-terminated strings

```
strcpy(buf1, buf2);
```
non-terminated strings

\texttt{\textbf{x}} \quad \texttt{\textbf{x}}

\begin{itemize}
  \item \texttt{strcpy(buf1, buf2);}
  \item \texttt{char buf1[4];
  \texttt{strncpy(buf1, buf2, 4);}}
\end{itemize}
non-terminated strings

\begin{center}
\begin{itemize}
  \item \textbf{strcpy} (\textit{buf1}, \textit{buf2});
  \item char \textit{buf1}[4]; \\
  \hspace{1cm} \textbf{strncpy} (\textit{buf1}, \textit{buf2}, 4);
\end{itemize}
\end{center}

The \textbf{strncpy()} function copies at most \textit{n} characters from \textit{s2} into \textit{s1}. If \textit{s2} is less than \textit{n} characters long, the remainder of \textit{s1} is filled with `\0' characters. Otherwise, \textit{s1} is not terminated.
non-terminated strings

```
char buf1[4];
strncpy(buf1, buf2, 4);
```

The `strncpy()` function copies at most `n` characters from `s2` into `s1`. If `s2` is less than `n` characters long, the remainder of `s1` is filled with `\0` characters. Otherwise, `s1` is not terminated.

```
char buf1[4] = TEST\0
char buf2[] = "TESTING"
```
non-terminated strings

```c
char buf1[4];
strncpy(buf1, buf2, 4);
```

The `strncpy()` function copies at most `n` characters from `s2` into `s1`. If `s2` is less than `n` characters long, the remainder of `s1` is filled with `\0` characters. Otherwise, `s1` is not terminated.

```
char buf1[4] = "TESTING";
```
non-terminated strings

Strcpy(buf1, buf2);

char buf1[4];
strncpy(buf1, buf2, 4);

The `strncpy()` function copies at most n characters from s2 into s1. If s2 is less than n characters long, the remainder of s1 is filled with `\0` characters. Otherwise, s1 is not terminated.

```
char buf1[4] = "TESTING"
char buf2[] = "TESTING"
```

printf("buf1 is [%s]\n",buf1);
non-terminated strings

```c
char buf1[4];
strncpy(buf1, buf2, 4);
```

The `strncpy()` function copies at most \( n \) characters from \( s_2 \) into \( s_1 \). If \( s_2 \) is less than \( n \) characters long, the remainder of \( s_1 \) is filled with `\0` characters. Otherwise, \( s_1 \) is not terminated.

<table>
<thead>
<tr>
<th>TEST</th>
<th>TESTING</th>
</tr>
</thead>
</table>

```c
char buf1[4]
char buf2[] = "TESTING"

printf("buf1 is [%s]\n",buf1);
```

$ buf1 is [TESTTESTING]
heap-unlink()
heap-unlink()
heap-unlink()
heap-unlink()
heap-unlink()
heap-unlink()
heap-unlink()
* Peter Szor - analysis of the slapper worm
int func(char *a, char *b) {
    char buf[12];
    strcpy(buf, a);
    strcpy(b, buf);
    return 1;
}
int func(char *a, char *b) {
    char buf[12];
    strcpy(buf, a);
    strcpy(b, buf);
    return 1;
}
so..

- Everything executable --> DEP
- DEP vs ret-2-libc (ROP)
SO..

- ASLR to beat ret-2-libc / ROP
- Single leaked / static address beats ASLR
- Partial Overwrites
- App specific..
so..

- ASLR based on stack libraries / ROP
- Single leaked / static address beats ASLR
- Partial Overwrites
- App specific..
So..

<table>
<thead>
<tr>
<th>Application</th>
<th>DEP (7)</th>
<th>DEP (XP)</th>
<th>Full ASLR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flash Player</td>
<td>N/A</td>
<td>N/A</td>
<td>YES</td>
</tr>
<tr>
<td>Sun Java JRE</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Adobe Reader</td>
<td><strong>YES</strong></td>
<td><strong>YES</strong></td>
<td>no</td>
</tr>
<tr>
<td>Mozilla Firefox</td>
<td><strong>YES</strong></td>
<td><strong>YES</strong></td>
<td>no</td>
</tr>
<tr>
<td>Apple Quicktime</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>VLC Media Player</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Apple iTunes</td>
<td><strong>YES</strong></td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Google Chrome</td>
<td><strong>YES</strong></td>
<td><strong>YES</strong></td>
<td><strong>YES</strong></td>
</tr>
<tr>
<td>Shockwave Player</td>
<td>N/A</td>
<td>N/A</td>
<td>no</td>
</tr>
<tr>
<td>OpenOffice.org</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Google Picasa</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Foxit Reader</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Opera</td>
<td><strong>YES</strong></td>
<td><strong>YES</strong></td>
<td>no</td>
</tr>
<tr>
<td>Winamp</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>RealPlayer</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Apple Safari</td>
<td><strong>YES</strong></td>
<td><strong>YES</strong></td>
<td>no</td>
</tr>
</tbody>
</table>

**DEP & ASLR (June 2010)**

Secunia - June 2010

- DEP without ASLR
- ASLR without DEP
- Without
Conclusions?

- What the ASLR/DEP taketh..
- The rich client side applications giveth back.
- Info. leakage attacks are an area of much research

http://ilm.thinkst.com/folklore/
Thanks!

- Marco Slaviero
- Brad Spengler (spender)
- PaX Team
- Halvar Flake
- icesurfer
- Nate Lawson
- Chris Wysopal
- Saumil Shah
- Matt Miller
- Ollie Whitehouse
- Dennis Groves
- Ivan Arce
- Mario Vilas
- Tyler Shields
- Dion Blazakis
- georgie
- Ben Nagy
- the Grugq.
- Bradley Cowie
- Barry Irwin
Questions?

http://ilm.thinkst.com/folklore

@haroonmeer
haroon@thinkst.com
http://blog.thinkst.com