Full Speed in Reverse
protecting legacy binaries from memory corruption attacks

Herbert Bos
VU University Amsterdam
Grants

• ERC StG “Rosetta”
• EU FP 7 Syssec
• DG Home iCode
CAN WE WIN?
Can we stop advanced malware?

• Like Stuxnet, Duqu, Zeus, TDL4, Flame(?)
Can we stop advanced malware?

- Zeus is a banking trojan that has been around since 2007
- Sold as a DIY toolkit, anyone can create their own botnet for $4000
- In 2010, the FBI discovered an organized crime network which stole over $70m using Zeus
Can we stop advanced malware?

Zeus v3

- In May 2011, the source code of Zeus v2 was leaked
- The leaked code evolved into a new Zeus P2P variant around October
- The Zeus P2P network is currently estimated to consist of 150,000 – 200,000 peers worldwide
Can we stop advanced malware?

• We tried
  – based on the work by Dennis Andriesss
Can we stop advanced malware?

• Plan:
  – Reverse engineer relevant parts
  – Poison the botnet
• Reversing is hard
  – several layers of encryption and obfuscation
• We dump the code *after* unpacking
  – and get 268,000 lines of assembly 😞
Can we stop advanced malware?

- By May 5th, a significant part of the peerlist entries in the Zeus network points to us.
Can we stop advanced malware?

- Zeus strikes back
  - DDoS
  - Update that blacklists our IP addresses
Can we stop advanced malware?

- We managed to take out 25% of the nodes
- In the end, we failed!
- Trivial to make even more resilient botnets
Perhaps we need better protection
Next few slides are based on

**Memory Errors: Past, Present, and Future (RAID’12)**

- Victor van der Veen, Nitish Sharma, Lorenzo Cavallaro, Herbert Bos
The most popular language in the world

[Bar chart showing popularity of various programming languages]

http://www.langpop.com/
Buffer overflows

• Perpetual top-3 threat
  – SANS CWE Top 25 Most dangerous programming errors

• Most drive-by-downloads
  – infect browser, download malware
Many defensive measures

- Canaries (StackGuard and friends)
- NX bit / $W \oplus X$
- ASLR
Evolution at work
Still they come
Vulnerabilities and exploits
(as percentage of total)
And legacy code?

- we do not have source code
  - we probably do not even have symbols
- we cannot recompile
  - most protective measures require recompilation
- we cannot protect
Taint Analysis?
Taint analysis

Windows
Argos
Linux

raise alarm when tainted bytes are loaded in PC
Taint tracking: useful, but slow
...and detects not the attack, but its manifestation.

just missed it!
...and does not detect attacks on non-control data at all!

```c
void get_private_medical_data (int uid) {
    int c, i = 0;
    int authorized = check(uid); // result=0 for attacker
    char patientid[8];

    printf("Type patientid, followed by the ");
    while (((c=getchar())!="#") patientid[i++] = c;

    if (authorized) print_medical_data (patientid);
    else printf("sorry, you are not authorized\n");
}
```

- trivially exploitable
- not prevented by ASLR, NX, or StackGuard
BinArmor
A Body Armour for Binaries
This talk is based on two papers

- Asia Slowinska, Traian Stancescu, Herbert Bos
  **Howard: a dynamic excavator for reverse engineering data structures** *(NDSS’11)*
- Asia Slowinska, Traian Stancescu, Herbert Bos
  **Body armor for binaries: preventing buffer overflows without recompilation** *(USENIX’12)*

[Images of Asia Slowinska and Traian Stancescu]
no source
no symbols
no clue?
In a nutshell...

(i) find arrays in binary programs
(ii) find accesses to arrays
(iii) rewrite the binary:
  - assign a color to each array
  - check colors on every array access

if a pointer that first pointed into an array...
...later accesses an area outside the array...

owned

crash()
In a nutshell...

(i) find arrays in binary programs

(ii) find accesses to arrays

(iii) rewrite the binary:
   - assign a color to each array
   - check colors on every array access

if a pointer that first pointed into an array...
...later accesses an area outside the array...

owned

crash()
In a nutshell...

(i) find arrays in binary programs
(ii) find accesses to arrays
(iii) rewrite the binary:
- assign a color to each array
- check colors on every array access

if a pointer that first pointed into an array...
...later accesses an area outside the array...

owned

syssec
In a nutshell...

1. Find arrays in binary programs
2. Find accesses to arrays
3. Rewrite the binary:
   - Assign a color to each array
   - Check colors on every array access

If a pointer that first pointed into an array...
...later accesses an area outside the array...

Owned

Crash()
Step 1: extract the arrays

Two possibilities

– symbol tables
– stripped

⇒ reverse engineering

let’s assume the latter
Problem
Why is it difficult?

```c
1. struct employee {
2.     char name[128];
3.     int year;
4.     int month;
5.     int day
6. };
7.
8. struct employee e;
9. e.year = 2010;
```
Why is it difficult?

1. `struct employee {
2.     char name[128];
3.     int year;
4.     int month;
5.     int day
6. };`
7.
8. `struct employee e;
9. e.year = 2010;

MISSING
• Data structures
Data structures: key insight

Yes, data is “apparently unstructured”
But usage is not!
Data structures: key insight

Yes, data is “apparently unstructured”
But usage is not!
Data structures: key insight

Yes, data is “apparently unstructured”
But usage is not!

Analyse dynamically

test
SE
inputs
app
DDE Emu
data structures
Intuition

• Observe how memory is used at runtime to detect data structures

• E.g., if A is a pointer
  1. and A is a function frame pointer, then *(A + 8) is perhaps a function argument
  2. and A is an address of a structure, then *(A + 8) is perhaps a field in this structure
  3. and A is an address of an array, then *(A + 8) is perhaps an element of this array
Approach

• Track pointers
  – find root pointers
  – track how pointers derive from each other
    • for any address $B = A + 8$, we need to know $A$.

• Challenges:
  – missing base pointers
    • for instance, a field of a `struct` on the stack may be updated using EBP rather than a pointer to the struct
  – multiple base pointers
    • e.g., normal access and `memset()`
Arrays are tricky

- Detection:
  - looks for chains of accesses in a loop
Arrays are tricky

- Detection:
  - looks for chains of accesses in a loop
Arrays are tricky

- Detection:
  - looks for chains of accesses in a loop
Arrays are tricky

- Detection:
  - looks for chains of accesses in a loop
  - and sets of accesses with same base in linear space
Interesting challenges

• Example:
  – Decide which accesses are relevant
    • Problems caused by e.g., memset-like functions

Reported by `memset`
Further Challenges

• Arrays
  – Nested loops
  – Consecutive loops
  – Boundary elements
  –
Further Challenges

• Arrays
  – Nested loops
  – Consecutive loops
  – Boundary elements
Further Challenges

• Arrays
  – Nested loops
  – Consecutive loops
  – Boundary elements
  – 
Further Challenges

• Arrays
  – Nested loops
  – Consecutive loops
  – Boundary elements
  –
Final mapping

- map access patterns to data structures
  - static memory: on program exit
  - heap memory: on free
  - stack frames: on return
Also: not everything is hidden
Key insight 2

Yes, data is “apparently unstructured”
But usage is not!

Usage (again) reveals semantics
Key insight 2

Yes, data is “apparently unstructured”
But usage is not!

Usage (again) reveals semantics
Semantics: key insights

- Yes, data is “apparently unstructured”
- But usage is not!
- Usage (again) reveals semantics
Key insight 3

Yes, data is “apparently unstructured”
But usage is not!

Propagate types from sources + sinks
Key insight 3

Yes, data is “apparently unstructured”
But usage is not!

Propagate types from sources + sinks
Key insight 3

Yes, data is “apparently unstructured”
But usage is not!

Propagate types from sources + sinks
Key insight 3

Yes, data is “apparently unstructured”
But usage is not!

Propagate types from sources + sinks
Yes, data is “apparently unstructured”
But usage is not!

Propagate types from sources + sinks
Key insight 3

Yes, data is “apparently unstructured”
But usage is not!

Propagate types from sources + sinks

open ("Herbert.doc", R_ONLY)
Key insight 3

Yes, data is “apparently unstructured”
But usage is not!

Propagate types from sources + sinks

open ("Herbert.doc", R_ONLY)
Results
# Results

Below is a table showing the size of different programs (in lines of code, LoC) and a bar chart illustrating the percentage of heap memory usage for these programs.

<table>
<thead>
<tr>
<th>Prog</th>
<th>LoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>wget</td>
<td>46K</td>
</tr>
<tr>
<td>fortune</td>
<td>2K</td>
</tr>
<tr>
<td>grep</td>
<td>24K</td>
</tr>
<tr>
<td>gzip</td>
<td>21K</td>
</tr>
<tr>
<td>lighttpd</td>
<td>21K</td>
</tr>
</tbody>
</table>

The bar chart shows how much memory is used by different types of variables (unused, flattened, unused, missed, ok) for each program.
Results

<table>
<thead>
<tr>
<th>Prog</th>
<th>LoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>wget</td>
<td>46K</td>
</tr>
<tr>
<td>fortune</td>
<td>2K</td>
</tr>
<tr>
<td>grep</td>
<td>24K</td>
</tr>
<tr>
<td>gzip</td>
<td>21K</td>
</tr>
<tr>
<td>lighttpd</td>
<td>21K</td>
</tr>
</tbody>
</table>

Heap Memory

- Unused arrays
- Flattened
- Unused
- Missed
- Ok

% of total

wget
lighttpd
gzip
grep
fortune
Results

<table>
<thead>
<tr>
<th>Prog</th>
<th>LoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>wget</td>
<td>46K</td>
</tr>
<tr>
<td>fortune</td>
<td>2K</td>
</tr>
<tr>
<td>grep</td>
<td>24K</td>
</tr>
<tr>
<td>gzip</td>
<td>21K</td>
</tr>
<tr>
<td>lighttpd</td>
<td>21K</td>
</tr>
</tbody>
</table>

Heap Memory

- unused arrays
- flattened
- unused
- missed
- ok
Results

<table>
<thead>
<tr>
<th>Prog</th>
<th>LoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>wget</td>
<td>46K</td>
</tr>
<tr>
<td>fortune</td>
<td>2K</td>
</tr>
<tr>
<td>grep</td>
<td>24K</td>
</tr>
<tr>
<td>gzip</td>
<td>21K</td>
</tr>
<tr>
<td>lighttpd</td>
<td>21K</td>
</tr>
</tbody>
</table>
Results

<table>
<thead>
<tr>
<th>Prog</th>
<th>LoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>wget</td>
<td>46K</td>
</tr>
<tr>
<td>fortune</td>
<td>2K</td>
</tr>
<tr>
<td>grep</td>
<td>24K</td>
</tr>
<tr>
<td>gzip</td>
<td>21K</td>
</tr>
<tr>
<td>lighttpd</td>
<td>21K</td>
</tr>
</tbody>
</table>
## Results

<table>
<thead>
<tr>
<th>Prog</th>
<th>LoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>wget</td>
<td>46K</td>
</tr>
<tr>
<td>fortune</td>
<td>2K</td>
</tr>
<tr>
<td>grep</td>
<td>24K</td>
</tr>
<tr>
<td>gzip</td>
<td>21K</td>
</tr>
<tr>
<td>lighttpd</td>
<td>21K</td>
</tr>
</tbody>
</table>

![Heap Memory Graph](image-url)
Results

<table>
<thead>
<tr>
<th>Prog</th>
<th>LoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>wget</td>
<td>46K</td>
</tr>
<tr>
<td>fortune</td>
<td>2K</td>
</tr>
<tr>
<td>grep</td>
<td>24K</td>
</tr>
<tr>
<td>gzip</td>
<td>21K</td>
</tr>
<tr>
<td>lighttpd</td>
<td>21K</td>
</tr>
</tbody>
</table>
Demo?
Step 2: find array accesses

In principle: very simple

- detect array accesses at runtime
- remember the instructions

Note: not complete
Step 3: rewrite the binary

- assign a color to each array
- check colors on every array access

if a pointer that first pointed into an array...

...later accesses an area outside the array...

(iii)

owned

crash()
Two Modes

• Protect at object level (like WIT, BBC)
  – given symbols: zero false positives

• Protect at subfield granularity (like no-one else)
  – no false positives seen in practice (but no guarantees)
THIS TALK

Focuses on the latter
A colourful protection

• give all arrays a unique colour

\[
p = \text{array}; \\
\text{ASSIGN pointer a colour} \\
col(p) = \text{RED} \\
i = 0; \\
\text{while}(!\text{stop}) \\
\{
    *(p + i) = 0; \\
    i++; \\
\}
\]
A colourful protection

- give all arrays a unique colour

```c
p = array;
ASSIGN pointer a colour
col(p) = RED
i = 0;
while(!stop)
{
    *(p + i) = 0;
    CHECK if colours match:
    mem_col(p+i) == col(p)?
    i++;
}
```
Reality requires subtle shades
Reality requires subtle shades

typedef struct pair {
    int x;
    int y;
} pair_t;

struct s {
    int age;
    pair_t buff[4];
    int privileged;
};

<table>
<thead>
<tr>
<th>privileged</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
</tr>
<tr>
<td>x</td>
</tr>
<tr>
<td>y</td>
</tr>
<tr>
<td>x</td>
</tr>
<tr>
<td>y</td>
</tr>
<tr>
<td>x</td>
</tr>
<tr>
<td>age</td>
</tr>
</tbody>
</table>

data in memory

C0

C1

buf

age

privileged
Reality requires subtle shades
In reality

Check: does the pointer colour match that of the location pointed to?
(left to right, in all shades, with blanks serving as wild cards)
Unfortunately, some code is colour blind!

```c
typedef struct pair {
    int x;
    int y;
} pair_t;

struct s {
    int age;
    pair_t buf[4];
    int privileged;
};

int *p;
for (p=objptr, p<sizeof(*objptr); p++) *p = 0;
```
So we mask some shades

/* initialize the buffer */
int *p;
tint len = 4; //buf length

for(p = mystruct.buf;
p < mystruct.buf+len;
p++)
{
    *p = 0;
}"
So we mask some shades

```c
/* initialize the buffer */
int *p;
int len = 4; // buf length
for(p = mystruct.buf;
p < mystruct.buf+len;
p++)
{
    *p = 0;
}
```

<table>
<thead>
<tr>
<th>Privilege</th>
<th>C_0</th>
<th>C_1</th>
<th>C_2</th>
<th>C_3</th>
<th>C_4</th>
</tr>
</thead>
<tbody>
<tr>
<td>age</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>buf[0]</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>buf[1]</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>buf[2]</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>buf[3]</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. With masks, all these colors match!
2. This access will fail
Performance?

Client applications

- gzip (1.6M)
- gzip (6.8M)
- gzip (67M)
- htget (any size)
- wget (any size)

Slowdown

Native performance

BodyArmor
Performance?

![Lighttpd response rate graph](image)

- **Native performance**
- **BodyArmor**

Slowdown:
- 1K: Native performance (1), BodyArmor (1.5)
- 10K: Native performance (2), BodyArmor (2.5)
- 100K: Native performance (3), BodyArmor (3.5)
- 1M: Native performance (2), BodyArmor (2.5)
- 10M: Native performance (1.5), BodyArmor (2)
Performance?

Nbench benchmark suite

overall: 2.9
# Effectiveness?

<table>
<thead>
<tr>
<th>Application</th>
<th>Type of vulnerability</th>
<th>Security advisory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proftpd 1.3.3a</td>
<td>Stack overflow</td>
<td>CVE-2010-4221</td>
</tr>
<tr>
<td>Htget 0.93 (1)</td>
<td>Stack overflow</td>
<td>CVE-2004-0852</td>
</tr>
<tr>
<td>Htget 0.93 (2)</td>
<td>Stack overflow</td>
<td>CVE-2004-0548</td>
</tr>
<tr>
<td>Aspell 0.50.5</td>
<td>Stack overflow</td>
<td>CVE-2003-0947</td>
</tr>
<tr>
<td>Iwconfig v.26</td>
<td>Stack overflow</td>
<td>CVE-2005-1019</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Application</th>
<th>Type of vulnerability</th>
<th>Security advisory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exim 4.41</td>
<td>Heap overflow, non-control data</td>
<td>CVE-2010-4344</td>
</tr>
<tr>
<td>bc-1.06 (1)</td>
<td>Heap overflow</td>
<td>Bugbench [27]</td>
</tr>
<tr>
<td>bc-1.06 (2)</td>
<td>Heap overflow</td>
<td>Bugbench [27]</td>
</tr>
<tr>
<td>Nullhttpd-0.5.1</td>
<td>Heap overflow, reproduced</td>
<td>CVE-2002-1496</td>
</tr>
<tr>
<td>Squid-2.3</td>
<td>Heap overflow, reproduced</td>
<td>Bugbench [27]</td>
</tr>
<tr>
<td>Ncompress 4.2.4</td>
<td>Stack overflow</td>
<td>CVE-2001-1413</td>
</tr>
</tbody>
</table>
Conclusions

• Memory errors
  – are not going to go away

• BinArmor
  – protect against attacks on non-control data
  – few (if any) FPs
  – expensive
  – not fully optimised yet!

http://www.cs.vu.nl/~herbertb/