Body Armour for Binaries

protecting legacy binaries from memory corruption attacks

Herbert Bos

VU University Amsterdam
Grants

- ERC StG “Rosetta”
- EU FP 7 Syssec
- DG Home iCode
Systems Security @ VU

- secure and reliable OS (ERC)
- reliable FS
- dynamic updates
- voting machines
- compiler techniques
- static analysis
- mobile phones
- decoupled security
- intrusion recovery
- memory corruption
- reverse engineering (ERC)
- high-speed monitoring
- payload execution
- code obfuscation
- security for legacy binaries
- attack + malware analysis
- malware collection
- honeypots (client+server)
- taint analysis
- dynamic analysis
- attack detection
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)*
The most popular language in the world

http://www.langpop.com/
The most popular language in the world

http://www.langpop.com/

http://www.google.com/codesearch
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⊕X
- ASLR

![Canaries](image.png)
Still they come
Memory Corruption as a Percentage of Total Reported

- % Vulnerabilities
- % Exploits

Vulnerabilities:
- 0%
- 10%
- 20%
- 30%
- 40%
- 50%

Exploits:
- 0%
- 10%
- 20%
- 30%
- 40%
- 50%

Year:
- 1996
- 1998
- 2000
- 2002
- 2004
- 2006
- 2008
- 2010
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 '!' key\n");
    while (((c=getchar())!='!') patentid[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
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

![](syssec)
In a nutshell...

(i) find arrays in binary programs
   - find accesses to arrays

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

(iii) 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...

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

```c
main()
{
    int x, y;
    for (;;)
        x = ...;
}
```
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;`
Why is it difficult?

MISSING
• Data structures
Data structures: key insight

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

Data structures: key insight

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

Propagate types from sources + sinks

open ("Herbert.doc", R_ONLY)
Yes, data is “apparently unstructured”
But usage is not!

Propagate types from sources + sinks

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

Diagram: tion of heap memory usage for different programs.
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>
# Results

## Heap Memory

<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>
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

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

(iii)

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

```c
p = array;
ASSIGN pointer a colour
col(p) = RED
i = 0;
while(!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
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

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

![Graph showing performance comparison between native and BodyArmor.

- Client applications
- Slowdown axis
- Bars for gzip (1.6M), gzip (6.8M), gzip (67M), htget (any size), wget (any size)

Legend:
- Native performance
- BodyArmor

The graph illustrates the performance slowdown for various client applications, comparing native performance to BodyArmor. The slowdown is shown across different file sizes and HTTP methods.
Performance?

![Graph showing Lighttpd response rate with bars for Native performance and BodyArmor, indicating performance comparison across different request sizes (1K, 10K, 100K, 1M, 10M).]
Performance?

![Nbench benchmark suite graph](image)

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

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

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