’T ain’t enough to fuzz
Heavy lifting

- Istvan Haller
- Asia Slowinska
- Erik Bosman
- Victor van der Veen
Some things we do

- Reverse engineering
- Exploitation/Attacks
- Binary protection
- Dynamic Analysis
- Obfuscation + deobfuscation
- Taint tracking
- Botnets
- Reliable OSs
- Argos

GreHack
2013-11-15
Today:

Buffer Overflows
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

• NX bit / DEP / W⊕X
• Canaries and Cookies
• ASLR
Still they come
Evolution at work

“Memory Errors: the Past, the Present and the Future” [RAID’12]

GreHack

2013-11-15
Vulnerabilities and exploits
(as percentage of total)

http://vvdveen.com/graphs/webpage.html
Nature of attacks
(stack-based overflows are getting rarer)

http://vvdveen.com/graphs/webpage.html
wouldn’t it be nice
if we found them
automatically
before release
Testing
Dowsing
A Guided Fuzzer to Find Vulnerabilities

Dowsing is a type of divination used to find ground water, buried treasure, rare gemstones, and now also bugs...
Timeline

Start
End of 2011
Summer’12
August’12
October’12
Jan ‘13
April’13

First results

NDDSS’12
REJECT

GreHack
2013-11-15
Where’s the fire?

• Buffer overflows are a top 3 threat!
  – Triggered under rare conditions

• Applications grow rapidly
  – Automated testing doesn’t scale!
Security testing today
Symbolic execution

• Example: let’s model the speed of a car

<table>
<thead>
<tr>
<th>Concrete values</th>
<th>Symbolic values</th>
</tr>
</thead>
<tbody>
<tr>
<td>115 km/h</td>
<td>100 ≤ v ≤ 120 km/h</td>
</tr>
<tr>
<td>115 km/h</td>
<td>0 ≤ v ≤ 120 km/h</td>
</tr>
<tr>
<td>250 km/h</td>
<td>v ≥ 0 km/h</td>
</tr>
</tbody>
</table>
Symbolic execution

if (a > 3)
    exit(0);

if (a > 2) {
    do_something0;
} else {
    if (a <= 5)
        do_something1;
    else
        assert(0);
}
Symbolic execution

if (a > 3)
    exit(0);

if (a > 2) {
    do_something0;
} else {
    if (a <= 5)
        do_something1;
    else
        assert(0);
}
Symbolic execution

```c
if (a > 3)
    exit(0);

if (a > 2) {
    do_something0;
} else {
    if (a <= 5)
        do_something1;
    else
        assert(0);
}
```
Symbolic execution

if (a > 3)
  exit(0);

if (a > 2) {
  do_something0;
} else {
  if (a <= 5)
    do_something1;
  else
    assert(0);
}
Symbolic execution

```c
if (a > 3)
    exit(0);

if (a > 2) {
    do_something0;
} else {
    if (a <= 5)
        do_something1;
    else
        assert(0);
}
```

```
\begin{tikzpicture}
  \node (a) at (0,0) {a};
  \node (a_gt_3) at (1,1) {a > 3};
  \node (a_le_3) at (-1,1) {a <= 3};
  \node (a_gt_2) at (1,0) {a > 2};
  \node (a_le_2) at (-1,0) {a <= 2};
  \draw (a) -- (a_gt_3);
  \draw (a) -- (a_le_3);
  \draw (a_gt_3) -- (a_gt_2);
  \draw (a_le_3) -- (a_le_2);
\end{tikzpicture}
```
Symbolic execution

```c
if (a > 3)
    exit(0);

if (a > 2) {
    do_something0;
} else {
    if (a <= 5)
        do_something1;
    else
        assert(0);
}
```

Diagram:
```
  a
 /  \
/    /
/     /
/      /
/       /
/       a <= 3
/       /  \
/       /    /
/       /     /
/       /      /
/       a <= 2
       /  \
      /    /
     /     /
    a <= 5
   /  \
  a > 5
```

GreHack
2013-11-15
Symbolic execution

```c
if (a > 3)
    exit(0);

if (a > 2) {
    do_something0;
} else {
    if (a <= 5)
        do_something1;
    else
        assert(0);
}
```

(a<=3) && (a<=2) && (a>5)
Symbolic execution

if (a > 3)
  exit(0);

if (a > 2) {
  do_something0;
} else {
  if (a <= 5)
    do_something1;
  else
    assert(0);
}
Symbolic execution

- Does not scale!
  - The number of states grows exponentially, so the analysis of a complex program can take ages!
  - E.g., nginx vulnerability not found within 8 hours
But we don’t want to test the entire program
Only the buggy bits!
Surely, bugs can be anywhere!

- Can they?
- What do we need for a buffer overflow?
  - Buffer
  - Accesses to that buffer
  - Loop
- We can look for these properties *a priori*!
Moreover...

- All loops are created equal, but some loops are more equal than others
  - Complex code is buggier than simple code
  - ...

GreHack
Buffer underrun in nginx

```c
while (p <= r->uri_end)
    switch (state)
    case sw_usual:  *u++ = ch;  ...  
    case sw_slash:   *u++ = ch;  ...  
      ...
    case sw_dot:     *u++ = ch;  ...  
        if (ch == '/') u--;  ...  
    case sw_dot_dot: *u++ = ch;  ...  
        if (ch == '/') u -= 4;  ...  
      ...
```

400 lines of code that make your head hurt
Idea: dowse for vulnerabilities

- Don’t try to verify all inputs
  - Focus the search for bugs on small and “potentially suspicious” code fragments

1 “looks bug-prone”

“Spot checking”
1. "looks bug-prone"
Identify places likely to have bugs

Buffer overflows in software

• Requirements:
  – An array
  – A pointer accessing the array
  – In a loop

• Find statically
  – Hundreds – thousands of loops

• Our strategy:
  – Analyze data flow graph
  – Rank based on complexity
How do we rank?

• We score based on
  – Instructions
  – Different constants
  – Pointer casts
  – ….
Does that work?! 

- Consider nginx...
  - 70% of loops have minimal complexity
  - Example loop is in the top 5%
1 “looks buggy”

2

PartA PartB PartC

3

PartA XXXX PartC
Input tracking

• Aim:
  – Infer relationships between inputs and candidates
  – Taint tracking

Peter and Dorothy Denning: tracking information flows since the 1970s!
Input tracking

Example: nginx HTTP request

Long input with multiple tokens.
GET /long/path/file HTTP/1.1
Host: thisisthehost.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 1337
Input tracking
Example: nginx HTTP request

```
GET /long/path/file HTTP/1.1
Host: thisisthehost.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 1337
```

⇒ Make only these bytes symbolic
1 “looks buggy”
Symbolic execution

Now possible?

Not quite, but getting close

More tricks are in the paper [USENIX SEC’13]
Symbolic execution

![Graph showing search time vs. symbolic input bytes]

- Depth First Search
- Code Coverage

PartA  XXXX  PartC
Our approach

![Diagram showing search time vs symbolic input bytes for Depth First Search, Code Coverage, and Value Coverage.](image-url)
## Results

<table>
<thead>
<tr>
<th>Program</th>
<th>Vulnerability</th>
<th><strong>Dowser</strong></th>
<th>Symbolic input</th>
</tr>
</thead>
<tbody>
<tr>
<td>nginx 0.6.32</td>
<td>CVE-2009-2629 heap underflow</td>
<td>253 sec</td>
<td>URI field 50 bytes</td>
</tr>
<tr>
<td>ffmpeg 0.5</td>
<td>UNKNOWN heap overread</td>
<td>48 sec</td>
<td>Huffman table 224 bytes</td>
</tr>
<tr>
<td>inspired 1.1.22</td>
<td>CVE-2012-1836 heap overflow</td>
<td>32 sec</td>
<td>DNS response 301 bytes</td>
</tr>
<tr>
<td>poppler 0.15.0</td>
<td>UNKNOWN heap overread</td>
<td>14 sec</td>
<td>JPEG image 1024 bytes</td>
</tr>
<tr>
<td>poppler 0.15.0</td>
<td>CVE-2010-3704 heap overflow</td>
<td>762 sec</td>
<td>Embedded font 1024 bytes</td>
</tr>
<tr>
<td>libexif 0.6.20</td>
<td>CVE-2012-2841 heap overflow</td>
<td>652 sec</td>
<td>EXIF tag/length 1024 + 4 bytes</td>
</tr>
<tr>
<td>libexif 0.6.20</td>
<td>CVE-2012-2840 off-by-one error</td>
<td>347 sec</td>
<td>EXIF tag/length 1024 + 4 bytes</td>
</tr>
<tr>
<td>libexif 0.6.20</td>
<td>CVE-2012-2813 heap overflow</td>
<td>277 sec</td>
<td>EXIF tag/length 1024 + 4 bytes</td>
</tr>
<tr>
<td>snort 2.4.0</td>
<td>CVE-2005-3252 stack overflow</td>
<td>617 sec</td>
<td>UDP packet 1100 bytes</td>
</tr>
</tbody>
</table>
So we found a buffer overflow

Now what?
How to make use of it?

• DEP makes direct execution of shellcode unlikely
• Instead: code reuse
  – Return to libc
  – ROP
NEW!

SIGRETURN ORIENTED PROGRAMMING
Powerlifting

• Erik Bosman
[deliberately left blank]

The SROP material has not been made public yet
Conclusions

• Memory corruption are here to stay
  – Good hunting ground for research topics
• Only scratching the surface of fuzzing
  – Dowsing looks promising
• Interesting to look at new defenses
  – CFI anyone?
• Shellcode, ROP, JOP, ...
  – Now SROP ➔ not the final word