Analysis and Defense of Vulnerabilities in Binary Code

David Brumley
Carnegie Mellon University
dbrumley@cmu.edu

Buggy programs are one of the leading causes of hacked computers

“Regularly Install Patches”
– Computer Security Wisdom

Patches Can Help Attackers
– Evil David

Evil David's Timeline

Evil David

Use Patch to Reverse Engineer Bug

T1

T2

Attack Unpatched Users

Evil David’s Timeline
Asia gets Patch Delay
N. America gets patched version P

Evil David's Timeline

I can reverse engineer an input demonstrating the bug in minutes for an important class of bugs

Input Validation Bugs
(Buffer Overflow, Format String, Integer Overflow, etc)

Patch-based Exploit Generation

Step 1: Get
A) Buggy B
B) Patched P

Step 2: Create Exploit

Step 3: Profit!
Exploits Can Help Defenses
- Good David

Exploits Demonstrate Bugs

Goal: Recognizer for unsafe inputs

Given: 1. Exploit 2. B

Given:

1. Exploit
2. B

Goal:
Recognizer for unsafe inputs

Evil David

All Inputs

Safe Inputs

B

Filter Recognizes Exploits

Goal: Recognizer for unsafe inputs

Given:

1. Exploit
2. B

Good David

Creates Filters to Defend Against Exploits

Filters are a core component of widely used defenses from Symantec, Trend Micro, etc.
Our Setting:
Buggy B and Patched P are Binary Programs

Talk Outline
1. Automatic Patch-Based Exploit Generation
   - Generate inputs that execute specific line of code (weakest precondition)
   - Results
2. Automatic Input Filter Generation
   - New program analysis approach to filter generation
   - Filters have accuracy guarantees
3. Vine Platform for Binary Analysis
   - Infrastructure
4. Questions and Answers

Vine: Security-Relevant Binary Program Analysis Architecture

- Binary code is everywhere
- Security of the code you run (not just the code compiled)
Running Example

- All integers unsigned
- 32-bits
- All arithmetic mod $2^{32}$
- B is binary code

```plaintext
read input
if input % 2 == 0
    s := input + 3
    s := input + 2
ptr := realloc(ptr, s)
```

Input $= 2^{32} - 2$

$2^{32} - 2 \mod 2 = 0$

$s := 0 (2^{32} - 2 + 2 \mod 2^{32})$

$ptr := realloc(ptr, 0)$

Using ptr is a problem

Wanted: $s > input$

Integer Overflow when: $\neg(s > input)$

Patch

```plaintext
if input % 2 == 0
    s := input + 3
    s := input + 2
ptr := realloc(ptr, s)
if s > input
    Error
```

```plaintext
if input % 2 == 0
    s := input + 3
    s := input + 2
ptr := realloc(ptr, s)
```
Exploits for B are inputs that fail new safety condition check in P
(s > input) = false

Exploit Generation
1. Diff B and P to identify location of new safety check
2. Create input that fails safety condition in P using Vine
3. Verify input is exploit on original buggy program B

Weakest Precondition (WP)
Most general condition on inputs to fail check

WP Technique:
Derive condition at step i-1 to execute line i
if input % 2==0
read input
s := input + 3
s := input + 2
if s > input
ptr := realloc(ptr, s)

WP Technique:
Derive condition
at step i-1 to
execute line i

WP C2
WP C1
Bad

WP C2
WP C1
Bad

if input % 2==0
read input
s := input + 3
s := input + 2
if s > input
ptr := realloc(ptr, s)

Condition Rule
When
if e then b1 else b2:
Compute:
C1 for b1
C2 for b2
Then:
C1 ⇒ C2, & ~e ⇒ C2

Substitution rule
When
x := e, C1
Then
C1 = C1[e/x]
C2 = ~(input + 3 % 2^32 > input)
C3 = ~(s > input)
P
if input % 2==0
read input
C1 =
C2 =
C3 =

Substitution rule
When
x := e, C1
Then
C1 = C1[e/x]
C2 = ~(input + 3 % 2^32 > input)
C3 = ~(s > input)
P
if input % 2==0
read input
C1 =
C2 =
C3 =
\[ C_3 = \]
\[
(input \% 2 == 0) \Rightarrow \\
\neg((input + 2 \% 2^{32}) > input) \\
\& \\
\neg((input \% 2 == 0) \Rightarrow \\
\neg((input + 3 \% 2^{32}) > input)
\]

**Condition Rule**

<table>
<thead>
<tr>
<th>When</th>
<th>C1 = e ( \Rightarrow ) C1 &amp; \neg e ( \Rightarrow ) C2</th>
</tr>
</thead>
</table>

**Loops**

- **Safe**
- **Bad**

Consider a fixed number of times

**Two Problems with Classic Weakest Precondition:**

1. Not suitable for programs with gotos (unstructured programs)
2. Final C is **exponential** in size
   - Due to substitution for assignment
   - Duplication of condition in branches

**Substitution rule**

\[ C_3 = \]
\[
(input \% 2 == 0) \Rightarrow \\
\neg((input + 2 \% 2^{32}) > input) \\
\& \\
\neg((input \% 2 == 0) \Rightarrow \\
\neg((input + 3 \% 2^{32}) > input)
\]

**Condition Rule**

Use solver to find inputs such that \( C_3(\text{input}) = \text{true} \)

\[ C_3 = \ldots \]
In Brumley et al 07:
1. Generalized WP for programs with goto's (unstructured programs)
2. Prove $O(n^2)$ size for condition C
   - $n$ = number of statements
   - Proof generalizes [Leino05]

Putting it All Together
1. Diff B and P to identify location of new safety check
2. Create input that fails safety condition
   A. Lift to Vine
   B. Calculate weakest precondition for vulnerability
   C. Use solver to find inputs that satisfy condition: $C(input) = true$
3. Verify input is exploit on original buggy program B

Exploit Generation Results

<table>
<thead>
<tr>
<th>Exploit</th>
<th>Type</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASPNet_Filter</td>
<td>Information Disclosure</td>
<td>29 sec</td>
</tr>
<tr>
<td>GDI</td>
<td>Hijack Control</td>
<td>119 sec</td>
</tr>
<tr>
<td>PNG</td>
<td>Hijack Control</td>
<td>131 sec</td>
</tr>
<tr>
<td>IE COMCTL32 (B)</td>
<td>Hijack Control</td>
<td>378 sec</td>
</tr>
<tr>
<td>IGMP</td>
<td>Denial of Service</td>
<td>186 sec</td>
</tr>
</tbody>
</table>

- No public exploit for 3 out of 5
- Exploit unique for other 2
When could technique fail?
- Cannot solve condition C
- Not enough loop iterations
- Timing vulnerabilities
- etc.

However, security must conservatively estimate attackers capabilities
(We don’t know for future bugs)

New Research Problem:
Prevent Patches From Helping Attackers

Ideas?
- Code Analysis: Obfuscate patches
  - Prevents diffing in our approach, no changes to current update schemes
  - Con: May slow down program, may be insufficient
- Crypto: Encrypt patch initially, broadcast decryption key
  - Fair: Everyone applies patch simultaneously
  - Con: Which patches to encrypt? Requires changes to current update schemes, offline hosts?
- Others

Talk Outline
1. Automatic Patch-Based Exploit Generation
   - Generate inputs that execute specific line of code (weakest precondition)
   - Results
2. Automatic Input Filter Generation
   - New program analysis approach to filter generation
   - Filters have accuracy guarantees
3. Vine Platform for Binary Analysis
   - Infrastructure
4. Questions and Answers
Goal: Generate Accurate Filter

Filtered Safe Input (False +)

Missed Exploit (False -)

Perfect Filter:
Recognizes exactly set of unsafe inputs

Previous Approaches: Generating Filter from Known Inputs
- Honeycomb [Hotnets03], Autograph [USENIXSEC04],
  Earlybird [OSDI04], Polygraph [IEEE S&P05],
  Hamsa [IEEE S&P06], Anagram [RAID 06], etc.

Previous Approaches:
Attacker Can Manipulate Filter Generator
- Attacker can fool the filter generator learning algorithm into making errors (thus F+ and F-) [VBS06]
  - Even if sophisticated learning algorithms are used
  - Even if all labels are accurate
  - Even if the algorithms is randomized
- Attackers can do this in practice
  [Purdue et al. 2005, Newsome et al. 2006]
Filter generation in practice?

Can we create filters with accuracy guarantees automatically? [Brumley 2005-present]

Filter?

Good David

Given

Single sample exploit

Infer safety condition from execution of sample exploit

Perfect Filter: Recognizer of Vulnerability Language [Brumley et al 06,07]

B

read input

if input % 2 = 0

s := input + 3

s := input + 2

ptr := realloc(ptr, s)

if s > input

Inferred from sample exploit (e.g., $2^{32} - 2$)
Q: Can we create filters with accuracy guarantees automatically? Yes!

Filter Optimization = Code Optimization
(requires binary analysis, not just binary rewriting)

Initial Filter

Optimized Filter

Filter Generation Results

<table>
<thead>
<tr>
<th>Name</th>
<th>Accuracy</th>
<th>Gen Time</th>
<th>Pct Buggy Program</th>
<th>Eval Time (μs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOBB 10</td>
<td>Perfect</td>
<td>1.5s</td>
<td>16%</td>
<td>13</td>
</tr>
<tr>
<td>COMCTL32</td>
<td>Perfect</td>
<td>1.2s</td>
<td>8%</td>
<td>31</td>
</tr>
<tr>
<td>MOBB 19</td>
<td>Perfect</td>
<td>9.4s</td>
<td>4%</td>
<td>6</td>
</tr>
</tbody>
</table>

Optimizations help program analysis

2x speedup for exploit generation by optimizing before WP

Exploits are inputs where \( \neg (\text{input} < 2^{32}-3) = \text{true} \)
Filters and techniques used by Symantec/Norton (> 2,000,000 install base)

Accuracy guaranteed by my work

Other Filters Based On Perfect Filter

Perfect Filter
- May be Turing Complete
- Sound Approximation
- Finite Execution, Boolean Predicate using WP [Brumley07]
- Filter over finite domain
- Exploits are inputs where ¬(input < 2^{23} - 3) = true
- Sound = No false +

Regular Expression Filter (e.g., Solve WP) [Brumley08]
- Fast, potentially many false negatives
Talk Outline
1. Automatic Patch-Based Exploit Generation
   - Generate inputs that execute specific line of code (weakest precondition)
   - Results
2. Automatic Input Filter Generation
   - New program analysis approach to filter generation
   - Filters have accuracy guarantees
3. Vine Platform for Binary Analysis
   - Infrastructure
4. Questions and Answers

Binary Code is Everywhere
Vine targets x86
But understanding x86 semantics is challenging:
- x86 is complex
- 100’s of instructions

- add a,b
- parity flag = ...
- carry flag = ...
- auxiliary carry flag = ...
- zero flag = ...
- signed flag = ...
- overflow flag = ...
- ... code for goto ...

Security of the code you run
- Binary analysis is different than source
  - Memory instead of buffers (mem[32-bit integer] = 8-bit value)
  - Jumps to middle of “functions”
  - Few types
  - Unstructured
- Vine targets analyzing programs with potentially few abstractions
  - Problems provide assumptions, not Vine

Vine
Faithful Binary Code Analysis
- Faithful, simplified, and explicit representation of binary code
- Vine Intermediate Language
  - Vine Principle: Distinguish what we know from what we don’t
  - a = a+b
  - goto L if carry
  - all control flow determined by flags
Security of the Code You Run

function foo() { … }

*fpp = &foo

addr of foo

call func at *fpp

foo() will be called if *fpp not reassigned, right?

User input changes called function!
Binary-Centric Approach: Vine makes it possible

- Binary code is everywhere
  - Vine addresses engineering challenges of x86
- Security of the code you run (not just the code compiled)
  - Low-level details matter
  - Vine is an infrastructure for developing and implementing binary program analyses
- Implementation:
  - 1.5 years old
  - Approx 16,500 lines C++ to raise to Vine
  - Approx 21,000 lines of OCaml for analysis & project
- 8 peer-reviewed pubs, 4 PhD Thesis, Research Partnerships with Industry (Symantec & Others) and Universities (U.Pitt & Berkeley)

Published Vine Projects

- Exploit Gen [IEEE08]
- Filter Gen [IEEE08, CSF07]
- Deviation Detection [Usenix sec07 best paper]
- Malware analysis [Yin et al]
- Reverse Engineer Protocols [Cabellero et al]
- Etc.

New Vine Projects

- Secure Patch Distribution
- Analysis-Resistant Malware
- Patch Correctness (e.g., for medical equipment, voting, etc.)
- Etc.

Research Contributions

- Automatic Patch-Based Exploit Generation
  - WP: Generate input that executes line of code
  - Delayed Patch Attacks are Practical
  - Current patch distribution schemes need to be redesigned
- Automatic filter generation
  - New approach that provides accuracy guarantees
  - Transitioning from research to practice
- Vine: Binary analysis techniques make it possible
  - Vine enables security-relevant program analysis
  - Fuels a variety of important research
- Additional research I haven’t told you about
  - Net security, Source-code analysis, How I broke RSA, etc.

Thank You

Questions?

Contact: dbrumley@cmu.edu
Related Work

• Bug Finding
  – Fuzzing, static bug finding, dynamic bug finding, etc.
  – Does not address our scenario
  – Techniques may be complementary

• Automatic test case generation
  – EXE [Cadar et al]
  – Dart [Godefroid et al]
  – May produce exponential size predicates, WP $O(n^2)$

• Binary Analysis
  – CodeSurfer/X86 [Reps & Balakrishnan 04-current]
  – Phoenix [Microsoft]

• Filter Generation
  – Work cited in talk
  – Bouncer/Vigilante [Costa et al SOSP05/07]
  – They use exponential algorithm, brumley07 is quadratic