Security through Multi-Layer Diversity

Meng Xu

(Qualifying Examination Presentation)
Bringing Diversity to Computing Monoculture

- Current computing monoculture leaves our infrastructure vulnerable to massive and rapid attacks.

- Knowing that victim systems run on a specific software stack, an attacker can compromise them deterministically.
Heartbleed bug 'will cost millions'

Revoking all SSL certificates leaked by Heartbleed will cost millions of dollars, according to Cloudflare, which provides services to website hosts.

Image: Codenomicon
Shellshock makes Heartbleed look insignificant

The new vulnerability in the Bash shell is the worst we’ve seen in many years. No software on critical systems can be assumed as safe.

By Larry Seltzer for Zero Day | September 29, 2014 — 11:59 GMT (04:59 PDT) | Topic: Cloud
Response from Security Community

- $W \oplus R$, ASLR, CFI, CPI, MPX
- Softbound, CETS
- Address Sanitizer, Memory Sanitizer, Thread Sanitizer
- ……
Limitations of Existing Schemes

Widely-deployed security schemes: $W \oplus R$, ASLR, CFI

→ Not hard to by-pass
Limitations of Existing Schemes

Widely-deployed security schemes: \(W \oplus R, ASLR, CFI\)
→ Not hard to by-pass

More sophisticated schemes: LLVM sanitizers
→ Offer protection against only specific vulnerabilities
→ Refuse to be combined due to conflicts in design
Limitations of Existing Schemes

Widely-deployed security schemes: $W \oplus R$, ASLR, CFI
  → Not hard to by-pass

More sophisticated schemes: LLVM sanitizers
  → Offer protection against only specific vulnerabilities
  → Refuse to be combined due to conflicts in design

Accumulated overhead: Softbound + CETS
  → 110% slowdown
A Biological Inspiration

Even the deadliest virus cannot kill all species because of gene diversity
Enhance System Security Through Diversity
Enhance System Security Through Diversity

Input

Software Stack

Output

Virtualization

Variant 1

Variant 2

Variant 3

Synchronize Execution & Consolidate Outputs

Output
Enhance System Security Through Diversity

Input (benign)

Virtualization

Variant 1
Variant 2
Variant 3

Synchronize Execution & Consolidate Outputs

Output (consensus)
Enhance System Security Through Diversity

Input (malicious)

Virtualization

Variant 1
Variant 2
Variant 3

Synchronize Execution & Consolidate Outputs

No output (divergence)
An attacker has to simultaneously compromise all variants in order to compromise the whole system.
Enhance System Security Through Diversity

Input

Zend

Process

Implementation

Linux

Platform

Software Stack

Output
Enhance System Security Through Diversity

Input

Virtualization

Process

ASan

MSan

UBSan

Zend

Linux

Variant 1

Zend

Linux

Variant 2

Zend

Linux

Variant 3

Synchronize Execution & Consolidate Outputs

Output
Enhance System Security Through Diversity

Implementation

Virtualization

Input

Zend

Linux

Software Stack

Output

Zend

Linux

Variant 1

Variant 2

Variant 3

Output

Synchronize Execution & Consolidate Outputs

ASan

MSan

UBSan

Zend

HHVM

JPHP
Enhance System Security Through Diversity

**Software Stack**
- Zend
- Linux

**Platform**
- Variants:
  - Variant 1: Zend (Linux)
  - Variant 2: HHVM (Windows)
  - Variant 3: JPHP (MacOS)

**Virtualization**
- Input
- ASan
- MSan
- UBSan

**Synchronize Execution & Consolidate Outputs**

**Output**
Enhance System Security Through Diversity

Virtualization

Input

Variant 1
Variant 2
Variant 3

Synchronize Execution & Consolidate Outputs

Output

Software Stack

Zend
Linux

PlatPal (Security’17)

Bunshin (ATC’17)

ASan
MSan
UBSan

Zend
Linux
Windows
MacOS

Zend
HHVM
JPHP

Future work

Input
Bunshin: Compositing Security Mechanisms through Diversification

Meng Xu, Kangjie Lu, Taesoo Kim, Wenke Lee

Georgia Tech

Presented at the 2017 USENIX Annual Technical Conference (ATC’17)
Battle against Memory Errors

Protect dangerous operation using **sanity checks**:
→ Auto-applied at compile time

```c
void foo(T *a) {
    *a = 0x1234;
}
```

```c
void foo(T *a) {
    if(!is_valid_address(a) {
        report_and_abort();
    }
    *a = 0x1234;
}
```
# Battle against Memory Errors

<table>
<thead>
<tr>
<th>Memory Error</th>
<th>Main Causes</th>
<th>Defenses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out-of-bound read/write</td>
<td>Lack of length check</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Integer overflow</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Format string bug</td>
<td>Softbound</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AddressSanitizer</td>
</tr>
<tr>
<td></td>
<td>Bad type casting</td>
<td></td>
</tr>
<tr>
<td>Use-after-free</td>
<td>Dangling pointer</td>
<td>CETS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AddressSanitizer</td>
</tr>
<tr>
<td></td>
<td>Double free</td>
<td></td>
</tr>
<tr>
<td>Uninitialized read</td>
<td>Lack of initialization</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Data structure alignment</td>
<td>MemorySanitizer</td>
</tr>
<tr>
<td></td>
<td>Subword copying</td>
<td></td>
</tr>
<tr>
<td>Undefined behaviors</td>
<td>Divide-by-zero</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Pointer misalignment</td>
<td>UndefinedBehaviorSanitizer</td>
</tr>
<tr>
<td></td>
<td>Null-pointer dereference</td>
<td></td>
</tr>
</tbody>
</table>
Comprehensive Protection with Bunshin

• Accumulated execution slowdown
  • Example: Softbound + CETS → 110% slowdown
  • Bunshin: Reduce to 60% or 40% (depends on the config)

• Implementation conflicts
  • Example: AddressSanitizer and MemorySanitizer
  • Bunshin: Seamlessly enforce conflicting sanitizers
Challenges for Bunshin

• How to generate these variants?
• What properties they should have?
• How to make them appear as one to outsiders?
• What is a “behavior” and what is a divergence?
• What if the sanitizers introduces new behaviors?
• Multi-threading support?
Variant Generation Principles

• Check distribution

• Sanitizer distribution
Check Distribution

Input

Virtualization

Partition 1
Variant 1

Partition 2
Variant 2

Partition 3
Variant 3

Synchronize Execution & Consolidate Outputs

Output
Sanitizer Distribution

Virtualization

Input

Variant 1
Variant 2
Variant 3

Address
Memory
Undef

Program

Input

Address
Memory
Undef

Variant 1
Synchronize Execution & Consolidate Outputs

Address
Memory
Undef

Variant 2

Address
Memory
Undef

Variant 3

Address
Memory
Undef

Output
Cost Profiling

- Calculate the slowdown caused by the sanity checks

```c
void foo(T *a) {
    timing_start();
    if(!is_valid_address(a) { report_and_abort();
    *a = 0x1234;
    timing_end();
}
```

```c
void foo(T *a) {
    timing_start();
    if(!is_valid_address(a) { report_and_abort();
    *a = 0x1234;
    timing_end();
}
```
Cost Distribution

• Equally distribute overhead to variants so that they execute at the same speed

<table>
<thead>
<tr>
<th>Variant 1</th>
<th>52% overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Foo</td>
<td>17%</td>
</tr>
<tr>
<td>Bar</td>
<td>28%</td>
</tr>
<tr>
<td>Baz</td>
<td>35%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Variant 2</th>
<th>48% overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bar</td>
<td>28%</td>
</tr>
<tr>
<td>Qux</td>
<td>20%</td>
</tr>
<tr>
<td>Baz</td>
<td>35%</td>
</tr>
<tr>
<td>Foo</td>
<td>17%</td>
</tr>
</tbody>
</table>
Variant Generation Process

- **Source code**
  - Security mechanisms (e.g., ASan, MSan, UBSan)

- **Variant generator**
  - Costs profiling
  - Overhead distribution
  - Variant compiling
    - "opt.

- **Variants**
  - "full"
    - w/ MSan
    - w/ ASan
  - "selective"
    - w/ UBSan
    - w/ ASan
    - ...
System Call Synchronization

Userspace

- Leader
- Partition 1
- Partition 2
- Partition 3
- Follower 1
- Follower 2

Kernel

- Syscall number
- Arguments
- Execution result
- Sync slot
System Call Synchronization

Userspace

<table>
<thead>
<tr>
<th>Partition 1</th>
<th>Leader</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Kernel

<table>
<thead>
<tr>
<th>Partition 2</th>
<th>Follower 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Partition 3</th>
<th>Follower 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

② Followers enter syscall

- Syscall number
- Arguments
- Execution result
- Sync slot
System Call Synchronization

Userspace: Leader

Kernel:
- Syscall number
- Arguments
- Execution result

Sync slot

③ Kernel execute the syscall only once
System Call Synchronization

Userspace
- Leader

Kernel
- Partition 1
- Partition 2
- Partition 3
- Leader fetches syscall result
- Followers fetch syscall result

Syscall number
Arguments
Execution result
sync slot
Strict and Selective Lockstep

Userspace

Partition 1
Leader

Partition 2
Follower 1

Partition 3
Follower 2

Kernel

Leader writes at the next available slot

sync ring buffer

Followers read at their own speed
Strict and Selective Lockstep

Userspace

Partition 1
Leader

Partition 2
Follower 1

Partition 3
Follower 2

Kernel
Always strictly synchronized for “write” related system calls

sync ring buffer
Multi-threading Support

Before fork

Original Execution group

After fork

New Execution group

Leader

New ring buffer

Follower 1

Follower 2
Multi-threading Support

Before fork

Original Execution group

After fork

New Execution group

Leader

New ring buffer

Follower 1

Follower 2

Works if there is no interleaving between threads
Multi-threading Support

Userspace

Leader

Follower 1

Follower 2

Kernel

Record

Enforce

Enforce

Total order of lock acquisition and releases
Multi-threading Support

Works under weak determinism (data race-free programs)

Implementation specific (pthread APIs only)

Total order of lock acquisition and releases
Evaluate Bunshin

- Robustness and Security
- Efficiency and Scalability
- Protection Distribution Case Studies
## Robustness

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Single/Multi-thread</th>
<th>Feature</th>
<th>Pass?</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPEC CPU2006</td>
<td>Single</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>SPLASH-2x</td>
<td>Multi</td>
<td>CPU Intensive</td>
<td>✓</td>
</tr>
<tr>
<td>PARSEC</td>
<td>Multi</td>
<td></td>
<td>✓ 6 out of 13</td>
</tr>
<tr>
<td>lighttpd</td>
<td>Single</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>nginx</td>
<td>Multi</td>
<td>I/O Intensive</td>
<td>✓</td>
</tr>
<tr>
<td>python, php</td>
<td>Single</td>
<td>Interpreter</td>
<td>✓</td>
</tr>
</tbody>
</table>
Security

• RIPE Benchmark

<table>
<thead>
<tr>
<th>Config</th>
<th>Succeed</th>
<th>Probabilistic</th>
<th>Failed</th>
<th>Not possible</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>114</td>
<td>16</td>
<td>720</td>
<td>2990</td>
</tr>
<tr>
<td>AddressSanitizer</td>
<td>8</td>
<td>0</td>
<td>842</td>
<td>2990</td>
</tr>
<tr>
<td>Bunshin</td>
<td>8</td>
<td>0</td>
<td>842</td>
<td>2990</td>
</tr>
</tbody>
</table>

• Real-world CVEs

<table>
<thead>
<tr>
<th>Config</th>
<th>CVE</th>
<th>Exploits</th>
<th>Sanitizer</th>
<th>Detect</th>
</tr>
</thead>
<tbody>
<tr>
<td>nginx-1.4.0</td>
<td>2013-2028</td>
<td>Blind ROP</td>
<td>AddressSanitizer</td>
<td>✓</td>
</tr>
<tr>
<td>cpython-2.7.10</td>
<td>2016-5636</td>
<td>Integer overflow</td>
<td>AddressSanitizer</td>
<td>✓</td>
</tr>
<tr>
<td>php-5.6.6</td>
<td>2015-4602</td>
<td>Type confusion</td>
<td>AddressSanitizer</td>
<td>✓</td>
</tr>
<tr>
<td>openssl-1.0.1a</td>
<td>2014-0160</td>
<td>Heartbleed</td>
<td>AddressSanitizer</td>
<td>✓</td>
</tr>
<tr>
<td>httpd-2.4.10</td>
<td>2014-3581</td>
<td>Null dereference</td>
<td>UndefinedBehaviorSanitizer</td>
<td>✓</td>
</tr>
</tbody>
</table>
## Performance

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Items</th>
<th>Strict-Lockstep</th>
<th>Selective-Lockstep</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SPEC CPU2006</strong> (19 Programs)</td>
<td>Max</td>
<td>17.5%</td>
<td>14.7%</td>
</tr>
<tr>
<td></td>
<td>Min</td>
<td>1.6%</td>
<td>1.0%</td>
</tr>
<tr>
<td></td>
<td>Ave</td>
<td>8.6%</td>
<td>5.6%</td>
</tr>
<tr>
<td><strong>SPLASH-2X / PARSEC</strong> (19 Programs)</td>
<td>Max</td>
<td>21.4%</td>
<td>18.9%</td>
</tr>
<tr>
<td></td>
<td>Min</td>
<td>10.7%</td>
<td>6.6%</td>
</tr>
<tr>
<td></td>
<td>Ave</td>
<td>16.6%</td>
<td>14.5%</td>
</tr>
<tr>
<td><strong>lighttpd</strong> 1MB File Request</td>
<td>Ave</td>
<td>1.44%</td>
<td>1.21%</td>
</tr>
<tr>
<td><strong>nginx</strong> 1MB File Request</td>
<td>Ave</td>
<td>1.71%</td>
<td>1.41%</td>
</tr>
</tbody>
</table>
Performance Highlights

- **Low** overhead (5% - 16%) for standard benchmarks
- **Negligible** overhead (<= 2%) for server programs
- Extra cost of ensuring weak determinism is **8%**
- Selective-lockstep saves around **3%** overhead
Scalability - Number of Variants

Number of variants

Sync Overhead (%)

Ave
Max
Min

2 4 6 8
1.7 4.4 11.2 20.9
0.5 6.6 10.5 11.4
0 0 0 0
37.6 17.2 11.4

Ave
Max
Min
Scalability - Number of Variants

The number of variants Bunshin can support with a reasonable overhead depends on machine configurations and program characteristics.
Scalability - System Load

Ave  | Max  | Min 
---|---|---
2%  | 0.2 | 0.2 
50% | 4.8 | 9.7 
99% | 6.6 | 13 

Number of variants

Sync Overhead (%)
Scalability - System Load

Bunshin works well in all levels of system load (i.e., Bunshin does not require exclusive cores)
Check Distribution - ASan

![Chart 1: Overhead (Whole, V1, V2, Bunshin)]

Overhead (%)

<table>
<thead>
<tr>
<th></th>
<th>Whole</th>
<th>V1</th>
<th>V2</th>
<th>Bunshin</th>
</tr>
</thead>
<tbody>
<tr>
<td>107</td>
<td>57.4</td>
<td>63</td>
<td>65.6</td>
<td>107</td>
</tr>
</tbody>
</table>

![Chart 2: Overhead (Whole, V1, V2, V3, Bunshin)]

Overhead (%)

<table>
<thead>
<tr>
<th></th>
<th>Whole</th>
<th>V1</th>
<th>V2</th>
<th>V3</th>
<th>Bunshin</th>
</tr>
</thead>
<tbody>
<tr>
<td>107</td>
<td>34.8</td>
<td>34.9</td>
<td>37.2</td>
<td>43.1</td>
<td>107</td>
</tr>
</tbody>
</table>
Sanitizer Distribution - UBSan

<table>
<thead>
<tr>
<th></th>
<th>Whole</th>
<th>V1</th>
<th>V2</th>
<th>Bunshin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overhead (%)</td>
<td>228</td>
<td>124</td>
<td>125</td>
<td>129</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Whole</th>
<th>V1</th>
<th>V2</th>
<th>V3</th>
<th>Bunshin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overhead (%)</td>
<td>228</td>
<td>77.2</td>
<td>78.7</td>
<td>88</td>
<td>94.5</td>
</tr>
</tbody>
</table>
Unifying LLVM Sanitizers

Overhead (%)

- gobmk
- povray
- h264ref
- average

- ASan
- MSan
- UBSan
- Bunshin

Overhead (%) values:
- gobmk: 116 (ASan), 158 (MSan), 165 (UBSan), 172 (Bunshin)
- povray: 141 (ASan), 205 (MSan), 189 (UBSan), 246 (Bunshin)
- h264ref: 112 (ASan), 191 (MSan), 207 (UBSan), 208 (Bunshin)
- average: 98.9 (ASan), 148 (MSan), 177 (UBSan), 172 (Bunshin)
Unifying LLVM Sanitizers

With an average of 5% more slowdown, Bunshin can seamlessly unify all three LLVM sanitizers.
Limitations and Future Work

- Finer-grained check distribution
- Sanitizer integration
- Record-and-replay
Conclusion

• It is feasible to achieve both comprehensive protection and high throughput with an N-version system

• Bunshin is effective in reducing slowdown caused by sanitizers
  • 107% $\rightarrow$ 47.1% for ASan, 228% $\rightarrow$ 94.5% for UBSan

• Bunshin can seamlessly unify three LLVM sanitizers with 5% extra slowdown

https://github.com/sslab-gatech/bunshin
(Source code will be released soon)
Enhance System Security Through Diversity

**Input**

**Virtualization**

**Output**

**Variant 1**

**Variant 2**

**Variant 3**

**Synchronize Execution & Consolidate Outputs**

**Zend**

**Linux**

**PlatPal (Security’17)**

**Bunshin (ATC’17)**

**Future work**

**ASan**

**MSan**

**UBSan**

**Zend**

**HHVM**

**JPHP**

**Linux**

**Windows**

**MacOS**

**Software Stack**

**Output**

Zend

Linux

PlatPal (Security’17)

Bunshin (ATC’17)
PlatPal: Detecting Malicious Documents with Platform Diversity

Meng Xu and Taesoo Kim

Georgia Tech

Presented at the 2017 USENIX Security Symposium (Security’17)
Malicious Documents On the Rise
APT28 Targets Hospitality Sector, Presents Threat to Travelers

August 11, 2017 | by Lindsay Smith, Ben Read | Threat Research

FireEye has moderate confidence that a campaign targeting the hospitality sector is attributed to Russian actor APT28. We believe this activity, which dates back to at least July 2017, was intended to target travelers to hotels throughout Europe and the Middle East. The actor has used several notable techniques in these incidents such as sniffing passwords from Wi-Fi traffic, poisoning the NetBIOS Name Service, and spreading laterally via the EternalBlue exploit.

APT28 Uses Malicious Document to Target Hospitality Industry
Microsoft PowerPoint exploit used to bypass antivirus and spread malware

It's the first time this exploit has been used to target PowerPoint users - and it's being used to distribute powerful Trojan malware, say researchers.

By Danny Palmer | August 14 2017 -- 16:49 GMT (17:49 BST) | Topic: Security
Adobe Components Exploited

Element parser

JavaScript engine

Font manager

System dependencies

137 CVEs in 2015

227 CVEs in 2016
Maldoc Formula

Flexibility of doc spec

+ A large attack surface

+ Less caution from users

= More opportunities to profit
# Battle against Maldoc - A Survey

<table>
<thead>
<tr>
<th>Category</th>
<th>Focus</th>
<th>Work</th>
<th>Year</th>
<th>Detection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td>JavaScript</td>
<td>PJScan</td>
<td>2011</td>
<td>Lexical analysis</td>
</tr>
<tr>
<td></td>
<td>JavaScript</td>
<td>Vatamanu et al.</td>
<td>2012</td>
<td>Token clustering</td>
</tr>
<tr>
<td></td>
<td>JavaScript</td>
<td>Lux0r</td>
<td>2014</td>
<td>API reference classification</td>
</tr>
<tr>
<td></td>
<td>JavaScript</td>
<td>MPScan</td>
<td>2013</td>
<td>Shellcode and opcode sig</td>
</tr>
<tr>
<td></td>
<td>Metadata</td>
<td>PDF Malware Slayer</td>
<td>2012</td>
<td>Linearized object path</td>
</tr>
<tr>
<td></td>
<td>Metadata</td>
<td>Srndic et al.</td>
<td>2013</td>
<td>Hierarchical structure</td>
</tr>
<tr>
<td></td>
<td>Metadata</td>
<td>PDFrate</td>
<td>2012</td>
<td>Content meta-features</td>
</tr>
<tr>
<td></td>
<td>Both</td>
<td>Maiorca et al.</td>
<td>2016</td>
<td>Many heuristics combined</td>
</tr>
<tr>
<td>Dynamic</td>
<td>JavaScript</td>
<td>MDScan</td>
<td>2011</td>
<td>Shellcode and opcode sig</td>
</tr>
<tr>
<td></td>
<td>JavaScript</td>
<td>PDF Scrutinizer</td>
<td>2012</td>
<td>Known attack patterns</td>
</tr>
<tr>
<td></td>
<td>JavaScript</td>
<td>ShellOS</td>
<td>2011</td>
<td>Memory access patterns</td>
</tr>
<tr>
<td></td>
<td>JavaScript</td>
<td>Liu et al.</td>
<td>2014</td>
<td>Common attack behaviors</td>
</tr>
<tr>
<td></td>
<td>Memory</td>
<td>CWXDetector</td>
<td>2012</td>
<td>Violation of invariants</td>
</tr>
</tbody>
</table>
# Reliance on External PDF Parser

<table>
<thead>
<tr>
<th>Category</th>
<th>Focus</th>
<th>Work</th>
<th>Year</th>
<th>Detection</th>
<th>External Parser ?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td>JavaScript</td>
<td>PJScan</td>
<td>2011</td>
<td>Lexical analysis</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>JavaScript</td>
<td>Vatamanu et al.</td>
<td>2012</td>
<td>Token clustering</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>JavaScript</td>
<td>Lux0r</td>
<td>2014</td>
<td>API reference classification</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>JavaScript</td>
<td>MPScan</td>
<td>2013</td>
<td>Shellcode and opcode sig</td>
<td>No</td>
</tr>
<tr>
<td>Metadata</td>
<td>PDF Malware Slayer</td>
<td>2012</td>
<td>Linearized object path</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Srndic et al.</td>
<td>2013</td>
<td>Hierarchical structure</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Metadata</td>
<td>PDFRate</td>
<td>2012</td>
<td>Content meta-features</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Both</td>
<td>Maiorca et al.</td>
<td>2016</td>
<td>Many heuristics combined</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Dynamic</td>
<td>JavaScript</td>
<td>MDScan</td>
<td>2011</td>
<td>Shellcode and opcode sig</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>JavaScript</td>
<td>PDF Scrutinizer</td>
<td>2012</td>
<td>Known attack patterns</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>JavaScript</td>
<td>ShellOS</td>
<td>2011</td>
<td>Memory access patterns</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Liu et al.</td>
<td>2014</td>
<td>Common attack behaviors</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Memory</td>
<td>CWXDetector</td>
<td>2012</td>
<td>Violation of invariants</td>
<td>No</td>
</tr>
</tbody>
</table>
## Reliance on External PDF Parser

<table>
<thead>
<tr>
<th>Category</th>
<th>Focus</th>
<th>Work</th>
<th>Year</th>
<th>Detection</th>
<th>External Parser ?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td>JavaScript</td>
<td>PJScan</td>
<td>2011</td>
<td>Lexical analysis</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>JavaScript</td>
<td>Vatamanu et al.</td>
<td>2012</td>
<td>Token clustering</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>JavaScript</td>
<td>Lux0r</td>
<td>2014</td>
<td>API reference classification</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>JavaScript</td>
<td>MPScan</td>
<td>2013</td>
<td>Shellcode and opcode sig</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Metadata</td>
<td>PDF Malware Slayer</td>
<td>2012</td>
<td>Linearized object path</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Metadata</td>
<td>Srndic et al.</td>
<td>2013</td>
<td>Shallcode and opcode sig</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Metadata</td>
<td>PDFRate</td>
<td>2012</td>
<td>Content meta-features</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Both</td>
<td>Maiorca et al.</td>
<td>2016</td>
<td>Many heuristics combined</td>
<td>Yes</td>
</tr>
<tr>
<td>Dynamic</td>
<td>JavaScript</td>
<td>MDScan</td>
<td>2011</td>
<td>Shellcode and opcode sig</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>JavaScript</td>
<td>PDF Scrutinizer</td>
<td>2012</td>
<td>Known attack patterns</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>JavaScript</td>
<td>ShellOS</td>
<td>2011</td>
<td>Memory access patterns</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>JavaScript</td>
<td>Liu et al.</td>
<td>2014</td>
<td>Common attack behaviors</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Memory</td>
<td>CWXDetector</td>
<td>2012</td>
<td>Violation of invariants</td>
<td>No</td>
</tr>
</tbody>
</table>

*Parser-confusion attacks*  
*(Carmony et al., NDSS’16)*
## Reliance on Machine Learning

<table>
<thead>
<tr>
<th>Category</th>
<th>Focus</th>
<th>Work</th>
<th>Year</th>
<th>Detection</th>
<th>Machine Learning ?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td>JavaScript</td>
<td>PJScan</td>
<td>2011</td>
<td>Lexical analysis</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>JavaScript</td>
<td>Vatamanu et al.</td>
<td>2012</td>
<td>Token clustering</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>JavaScript</td>
<td>Lux0r</td>
<td>2014</td>
<td>API reference classification</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>JavaScript</td>
<td>MPScan</td>
<td>2013</td>
<td>Shellcode and opcode sig</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Metadata</td>
<td>PDF Malware Slayer</td>
<td>2012</td>
<td>Linearized object path</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Metadata</td>
<td>Srndic et al.</td>
<td>2013</td>
<td>Hierarchical structure</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Metadata</td>
<td>PDFrate</td>
<td>2012</td>
<td>Content meta-features</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Both</td>
<td>Maiorca et al.</td>
<td>2016</td>
<td>Many heuristics combined</td>
<td>Yes</td>
</tr>
<tr>
<td>Dynamic</td>
<td>JavaScript</td>
<td>MDScan</td>
<td>2011</td>
<td>Shellcode and opcode sig</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>JavaScript</td>
<td>PDF Scrutinizer</td>
<td>2012</td>
<td>Known attack patterns</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>JavaScript</td>
<td>ShellOS</td>
<td>2011</td>
<td>Memory access patterns</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>JavaScript</td>
<td>Liu et al.</td>
<td>2014</td>
<td>Common attack behaviors</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Memory</td>
<td>CWXDetector</td>
<td>2012</td>
<td>Violation of invariants</td>
<td>No</td>
</tr>
</tbody>
</table>
## Reliance on Machine Learning

<table>
<thead>
<tr>
<th>Category</th>
<th>Focus</th>
<th>Work</th>
<th>Year</th>
<th>Detection</th>
<th>Machine Learning ?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td>JavaScript</td>
<td>PJScan</td>
<td>2011</td>
<td>Lexical analysis</td>
<td>Yes</td>
</tr>
<tr>
<td>Static</td>
<td>JavaScript</td>
<td>Vatamanu et al.</td>
<td>2012</td>
<td>Token clustering</td>
<td>Yes</td>
</tr>
<tr>
<td>Static</td>
<td>JavaScript</td>
<td>Lux0r</td>
<td>2014</td>
<td>API reference classification</td>
<td>Yes</td>
</tr>
<tr>
<td>Static</td>
<td>JavaScript</td>
<td>MPScan</td>
<td>2013</td>
<td>Shellcode and opcode sig</td>
<td>No</td>
</tr>
<tr>
<td>Static</td>
<td>Metadata</td>
<td>PDF Malware Slayer</td>
<td>2012</td>
<td>Linearized object path</td>
<td>Yes</td>
</tr>
<tr>
<td>Static</td>
<td>Metadata</td>
<td>Srndic et al.</td>
<td>2012</td>
<td>Automatic classifier evasions</td>
<td>Yes</td>
</tr>
<tr>
<td>Static</td>
<td>Metadata</td>
<td>PDFrate</td>
<td>2012</td>
<td>Content meta-features</td>
<td>Yes</td>
</tr>
<tr>
<td>Static</td>
<td>Both</td>
<td>Maiorca et al.</td>
<td>2016</td>
<td>Many heuristics combined</td>
<td>Yes</td>
</tr>
<tr>
<td>Dynamic</td>
<td>JavaScript</td>
<td>MDScan</td>
<td>2011</td>
<td>Shellcode and opcode sig</td>
<td>No</td>
</tr>
<tr>
<td>Dynamic</td>
<td>JavaScript</td>
<td>PDF Scrutinizer</td>
<td>2012</td>
<td>Known attack patterns</td>
<td>No</td>
</tr>
<tr>
<td>Dynamic</td>
<td>JavaScript</td>
<td>ShellOS</td>
<td>2011</td>
<td>Memory access patterns</td>
<td>No</td>
</tr>
<tr>
<td>Dynamic</td>
<td>JavaScript</td>
<td>Liu et al.</td>
<td>2014</td>
<td>Common attack behaviors</td>
<td>No</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Memory</td>
<td>CWXDetector</td>
<td>2012</td>
<td>Violation of invariants</td>
<td>No</td>
</tr>
</tbody>
</table>
## Reliance on Known Attacks

<table>
<thead>
<tr>
<th>Category</th>
<th>Focus</th>
<th>Work</th>
<th>Year</th>
<th>Detection</th>
<th>Known Attacks ?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td>JavaScript</td>
<td>PJScan</td>
<td>2011</td>
<td>Lexical analysis</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>JavaScript</td>
<td>Vatamanu et al.</td>
<td>2012</td>
<td>Token clustering</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>JavaScript</td>
<td>Lux0r</td>
<td>2014</td>
<td>API reference classification</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>JavaScript</td>
<td>MPScan</td>
<td>2013</td>
<td>Shellcode and opcode sig</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Metadata</td>
<td>PDF Malware Slayer</td>
<td>2012</td>
<td>Linearized object path</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Metadata</td>
<td>Srndic et al.</td>
<td>2013</td>
<td>Hierarchical structure</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Metadata</td>
<td>PDFrate</td>
<td>2012</td>
<td>Content meta-features</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Both</td>
<td>Maiorca et al.</td>
<td>2016</td>
<td>Many heuristics combined</td>
<td>Yes</td>
</tr>
<tr>
<td>Dynamic</td>
<td>JavaScript</td>
<td>MDScan</td>
<td>2011</td>
<td>Shellcode and opcode sig</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>JavaScript</td>
<td>PDF Scrutinizer</td>
<td>2012</td>
<td>Known attack patterns</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>JavaScript</td>
<td>ShellOS</td>
<td>2011</td>
<td>Memory access patterns</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>JavaScript</td>
<td>Liu et al.</td>
<td>2014</td>
<td>Common attack behaviors</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Memory</td>
<td>CWXDetector</td>
<td>2012</td>
<td>Violation of invariants</td>
<td>No</td>
</tr>
</tbody>
</table>
## Reliance on Known Attacks

<table>
<thead>
<tr>
<th>Category</th>
<th>Focus</th>
<th>Work</th>
<th>Year</th>
<th>Detection</th>
<th>Known Attacks ?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td>JavaScript</td>
<td>PJScan</td>
<td>2011</td>
<td>Lexical analysis</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>JavaScript</td>
<td>Vatamanu et al.</td>
<td>2012</td>
<td>Token clustering</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>JavaScript</td>
<td>Lux0r</td>
<td>2014</td>
<td>API reference classification</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>JavaScript</td>
<td>MPScan</td>
<td>2013</td>
<td>Shellcode and opcode sig</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Metadata</td>
<td>PDF Malware Slayer</td>
<td>2012</td>
<td>Linearized object path</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Metadata</td>
<td>Srndic et al.</td>
<td>2013</td>
<td>Hierarchical structure</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Metadata</td>
<td>PDFrate</td>
<td>2012</td>
<td>Content meta-features</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Both</td>
<td>Maiorca et al.</td>
<td>2016</td>
<td>Many heuristics combined</td>
<td>Yes</td>
</tr>
<tr>
<td>Dynamic</td>
<td>JavaScript</td>
<td>MDScan</td>
<td>2011</td>
<td>Shellcode and opcode sig</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>JavaScript</td>
<td>PDF Scrutinizer</td>
<td>2012</td>
<td>Known attack patterns</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>JavaScript</td>
<td>ShellOS</td>
<td>2011</td>
<td>Memory access patterns</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>JavaScript</td>
<td>Liu et al.</td>
<td>2014</td>
<td>Common attack behaviors</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Memory</td>
<td>CWXDetector</td>
<td>2012</td>
<td>Violation of invariants</td>
<td>No</td>
</tr>
</tbody>
</table>

---

How about zero-day attacks?
## Reliance on Detectable Discrepancy
(between benign and malicious docs)

<table>
<thead>
<tr>
<th>Category</th>
<th>Focus</th>
<th>Work</th>
<th>Year</th>
<th>Detection</th>
<th>Discrepancy?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td>JavaScript</td>
<td>PJScan</td>
<td>2011</td>
<td>Lexical analysis</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>JavaScript</td>
<td>Vatamanu et al.</td>
<td>2012</td>
<td>Token clustering</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>JavaScript</td>
<td>Lux0r</td>
<td>2014</td>
<td>API reference classification</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>JavaScript</td>
<td>MPScan</td>
<td>2013</td>
<td>Shellcode and opcode sig</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Metadata</td>
<td>PDF Malware Slayer</td>
<td>2012</td>
<td>Linearized object path</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Metadata</td>
<td>Srndic et al.</td>
<td>2013</td>
<td>Hierarchical structure</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Metadata</td>
<td>PDFrate</td>
<td>2012</td>
<td>Content meta-features</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Both</td>
<td>Maiorca et al.</td>
<td>2016</td>
<td>Many heuristics combined</td>
<td>Yes</td>
</tr>
<tr>
<td>Dynamic</td>
<td>JavaScript</td>
<td>MDScan</td>
<td>2011</td>
<td>Shellcode and opcode sig</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>JavaScript</td>
<td>PDF Scrutinizer</td>
<td>2012</td>
<td>Known attack patterns</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>JavaScript</td>
<td>ShellOS</td>
<td>2011</td>
<td>Memory access patterns</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>JavaScript</td>
<td>Liu et al.</td>
<td>2014</td>
<td>Common attack behaviors</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Memory</td>
<td>CWXDetector</td>
<td>2012</td>
<td>Violation of invariants</td>
<td>No</td>
</tr>
</tbody>
</table>
## Reliance on Detectable Discrepancy
(between benign and malicious docs)

<table>
<thead>
<tr>
<th>Category</th>
<th>Focus</th>
<th>Work</th>
<th>Year</th>
<th>Detection</th>
<th>Discrepancy ?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td>JavaScript</td>
<td>PJScan</td>
<td>2011</td>
<td>Lexical analysis</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>JavaScript</td>
<td>Vatamanu et al.</td>
<td>2012</td>
<td>Token clustering</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>JavaScript</td>
<td>Lux0r</td>
<td>2014</td>
<td>API reference classification</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>JavaScript</td>
<td>MPScan</td>
<td>2013</td>
<td>Shellcode and opcode sig</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Metadata</td>
<td>PDF Malware Slayer</td>
<td>2012</td>
<td>Linearized object path</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Metadata</td>
<td>Srndic et al.</td>
<td>2013</td>
<td>Hierarchical structure</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Metadata</td>
<td>PDFrate</td>
<td>2012</td>
<td>Content meta-features</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Both</td>
<td>Maiorca et al.</td>
<td>2016</td>
<td>Many heuristics combined</td>
<td>Yes</td>
</tr>
<tr>
<td>Dynamic</td>
<td>JavaScript</td>
<td>MDScan</td>
<td>2011</td>
<td>Shellcode and opcode sig</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>JavaScript</td>
<td>PDF Scrutinizer</td>
<td>2012</td>
<td>Known attack patterns</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>JavaScript</td>
<td>ShellOS</td>
<td>2011</td>
<td>Memory access patterns</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>JavaScript</td>
<td>Liu et al.</td>
<td>2014</td>
<td>Common attack behaviors</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Memory</td>
<td>CWXDetector</td>
<td>2012</td>
<td>Violation of invariants</td>
<td>No</td>
</tr>
</tbody>
</table>

**Mimicry and reverse mimicry attacks**
*(Srndic et al., Oakland’14 and Maiorca et al, AsiaCCS’13)*
Highlights of the Survey

Prior works rely on

• External PDF parsers  →  Parser-confusion attacks
• Machine learning   →  Automatic classifier evasion
• Known attack signatures  →  Zero-day attacks
• Detectable discrepancy →  Mimicry and reverse mimicry
Motivations for PlatPal

Prior works rely on

• External PDF parsers
• Machine learning
• Known attack signatures
• Detectable discrepancy

What PlatPal aims to achieve

• Using Adobe’s parser
• Using only simple heuristics
• Capable to detect zero-days
• Do not assume discrepancy
• Complementary to prior works
Motivations for PlatPal

Prior works rely on

- External PDF parsers
- Machine learning
- Known attack signatures
- Detectable discrepancy

What PlatPal aims to achieve

- Using Adobe’s parser
- Capable to detect zero-days
- Do not assume discrepancy
Motivations for PlatPal

Prior works rely on

- External PDF parsers
- Machine learning
- Known attack signatures
- Detectable discrepancy

What PlatPal aims to achieve

- Using Adobe’s parser
- Using only simple heuristics
Motivations for PlatPal

Prior works rely on

- External PDF parsers
- Machine learning
- Known attack signatures
- Detectable discrepancy

What PlatPal aims to achieve

- Using Adobe’s parser
- Using only simple heuristics
- Capable to detect zero-days
Motivations for PlatPal

<table>
<thead>
<tr>
<th>Prior works rely on</th>
<th>What PlatPal aims to achieve</th>
</tr>
</thead>
<tbody>
<tr>
<td>• External PDF parsers</td>
<td>• Using Adobe’s parser</td>
</tr>
<tr>
<td>• Machine learning</td>
<td>• Using only simple heuristics</td>
</tr>
<tr>
<td>• Known attack signatures</td>
<td>• Capable to detect zero-days</td>
</tr>
<tr>
<td>• Detectable discrepancy</td>
<td>• Do not assume discrepancy</td>
</tr>
</tbody>
</table>
Motivations for PlatPal

Prior works rely on

• External PDF parsers
• Machine learning
• Known attack signatures
• Detectable discrepancy

What PlatPal aims to achieve

• Using Adobe’s parser
• Using only simple heuristics
• Capable to detect zero-days
• Do not assume discrepancy
• Complementary to prior works
A Motivating Example

• A CVE-2013-2729 PoC against Adobe Reader 10.1.4

SHA-1: 74543610d9908698cb0b4bfc73fc007bfeb6d84
Platform Diversity as A Heuristic

When the same document is opened across different platforms:

- A **benign** document “behaves” the **same**
- A **malicious** document “behaves” **differently**
Questions for PlatPal

• What is a “behavior” ?
• What is a divergence ?
• How to trace them ?
• How to compare them ?
PlatPal Basic Setup

Virtual Machine 1

Adobe Reader

Windows Host

Virtual Machine 2

Adobe Reader

MacOS Host
PlatPal Dual-Level Tracing

Virtual Machine 1

Adobe Reader
Internal Tracer

Traces of PDF processing

Adobe Reader
Internal Tracer

Virtual Machine 2

Windows Host

MacOS Host
PlatPal Dual-Level Tracing

Virtual Machine 1

Adobe Reader

Internal Tracer

Syscalls

External Tracer

Impacts on host platform

Windows Host

Virtual Machine 2

Adobe Reader

Internal Tracer

Syscalls

External Tracer

Impacts on host platform

MacOS Host

Traces of PDF processing

PDF

Adobe

?
PlatPal Internal Tracer

- Implemented as an Adobe Reader plugin.
- Hooks critical functions and callbacks during the PDF processing lifecycle.
- Very fast and stable across Adobe Reader versions.
PlatPal External Tracer

- Implemented based on *NtTrace* (for Windows) and *Dtrace* (for MacOS).
- Resembles high-level system impacts in the same manner as Cuckoo guest agent.
- Starts tracing only after the document is loaded into Adobe Reader.
PlatPal Automated Workflow
Evaluate PlatPal

- Robustness against benign samples
  - A benign document “behaves” the same?

- Effectiveness against malicious samples
  - A malicious document “behaves” differently?

- Speed and resource usages
Robustness

- 1000 samples from Google search.
- 30 samples that use advanced features in PDF standards from PDF learning sites.

<table>
<thead>
<tr>
<th>Sample Type</th>
<th>Number of Samples</th>
<th>Divergence Detected? (i.e., False Positive)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plain PDF</td>
<td>966</td>
<td>No</td>
</tr>
<tr>
<td>Embedded fonts</td>
<td>34</td>
<td>No</td>
</tr>
<tr>
<td>JavaScript code</td>
<td>32</td>
<td>No</td>
</tr>
<tr>
<td>AcroForm</td>
<td>17</td>
<td>No</td>
</tr>
<tr>
<td>3D objects</td>
<td>2</td>
<td>No</td>
</tr>
</tbody>
</table>
Effectiveness

- 320 malicious samples from VirusTotal with CVE labels.
- Restricted to analyze CVEs published after 2013.
- Use the most recent version of Adobe Reader when the CVE is published.
Effectiveness

Analysis Results of 320 Maldoc Samples

- No Divergence: 65%
- Both Crash: 11%
- Divergence: 24%
Effectiveness

Analysis Results of 320 Maldoc Samples

- No Divergence: 24%

Breakdown of 77 potentially false positives:

- Targets old versions: 25%
- Mis-classified by AV vendor: 47%
- No malicious activity triggered: 26%
- Unknown: 3%
## Time and Resource Usages

### Average Analysis Time Breakdown (unit. Seconds)

<table>
<thead>
<tr>
<th>Item</th>
<th>Windows</th>
<th>MacOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Snapshot restore</td>
<td>9.7</td>
<td>12.6</td>
</tr>
<tr>
<td>Document parsing</td>
<td>0.5</td>
<td>0.6</td>
</tr>
<tr>
<td>Script execution</td>
<td>10.5</td>
<td>5.1</td>
</tr>
<tr>
<td>Element rendering</td>
<td>7.3</td>
<td>6.2</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>23.7</strong></td>
<td><strong>22.1</strong></td>
</tr>
</tbody>
</table>

### Resource Usages

- 2GB memory per running virtual machine.
- 60GB disk space for Windows and MacOS snapshots that each corresponds to one of the 6 Adobe Readers versions.
Evaluation Highlights

• Confirms our fundamental assumption in general:
  - **benign** document “behaves” the **same**
  - **malicious** document “behaves” **differently**

• PlatPal is subject to the pitfalls of dynamic analysis
  - i.e., prepare the environment to lure the malicious behaviors

• Incurs reasonable analysis time to make PlatPal practical
Further Analysis

• What could be the root causes of these divergences?
## Diversified Factors across Platforms

<table>
<thead>
<tr>
<th>Category</th>
<th>Factor</th>
<th>Windows</th>
<th>MacOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shellcode Creation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory Management</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Platform Features</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Diversified Factors across Platforms

<table>
<thead>
<tr>
<th>Category</th>
<th>Factor</th>
<th>Windows</th>
<th>MacOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shellcode Creation</td>
<td>Syscall semantics</td>
<td>Both the syscall number and the register set used to hold syscall arguments are different</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Calling convention</td>
<td><em>rcx, rdx, r8</em> for first 3 args</td>
<td><em>rdi, rsi, rdx</em> for first 3 args</td>
</tr>
<tr>
<td></td>
<td>Library dependencies</td>
<td>e.g., <em>LoadLibraryA</em></td>
<td>e.g. <em>dlopen</em></td>
</tr>
<tr>
<td>Memory Management</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Platform Features</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Diversified Factors across Platforms

<table>
<thead>
<tr>
<th>Category</th>
<th>Factor</th>
<th>Windows</th>
<th>MacOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shellcode Creation</td>
<td>Syscall semantics</td>
<td>Both the syscall number and the register set used to hold syscall arguments are different</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Calling convention</td>
<td><em>rcx, rdx, r8</em> for first 3 args</td>
<td><em>rdi, rsi, rdx</em> for first 3 args</td>
</tr>
<tr>
<td></td>
<td>Library dependencies</td>
<td><em>e.g., LoadLibraryA</em></td>
<td><em>e.g. dlopen</em></td>
</tr>
<tr>
<td>Memory Management</td>
<td>Memory layout</td>
<td>Offset from attack point (e.g., overflowed buffer) to target address (e.g., vtable entries) are different</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Heap management</td>
<td>Segment heap</td>
<td>Magazine malloc</td>
</tr>
<tr>
<td>Platform Features</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Diversified Factors across Platforms

<table>
<thead>
<tr>
<th>Category</th>
<th>Factor</th>
<th>Windows</th>
<th>MacOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shellcode Creation</td>
<td>Syscall semantics</td>
<td>Both the syscall number and the register set used to hold syscall arguments are different</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Calling convention</td>
<td><em>rcx, rdx, r8</em> for first 3 args</td>
<td><em>rdi, rsi, rdx</em> for first 3 args</td>
</tr>
<tr>
<td></td>
<td>Library dependencies</td>
<td>e.g., <em>LoadLibraryA</em></td>
<td>e.g. <em>dlopen</em></td>
</tr>
<tr>
<td>Memory Management</td>
<td>Memory layout</td>
<td>Offset from attack point (e.g., overflowed buffer) to target address (e.g., vtable entries) are different</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Heap management</td>
<td>Segment heap</td>
<td>Magazine malloc</td>
</tr>
<tr>
<td>Platform Features</td>
<td>Executable format</td>
<td>COM, PE, NE</td>
<td>Mach-O</td>
</tr>
<tr>
<td></td>
<td>Filesystem semantics</td>
<td>\ as separator, prefixed drive letter C:\</td>
<td>/ as separator, no prefixed drive letter</td>
</tr>
<tr>
<td></td>
<td>Config and info hub</td>
<td>registry</td>
<td>proc</td>
</tr>
<tr>
<td></td>
<td>Expected programs</td>
<td>MS Office, IE, etc</td>
<td>Safari, etc</td>
</tr>
</tbody>
</table>
Back to The Motivating Example

1. Allocate 1000 300-bytes chunks

2. Free 1 in every 10

3. Load a 300-byte malicious BMP image

4. Corrupt heap metadata due to a buffer overflow

5. Free BMP image, but what is actually freed is slot 9

6. A vtable of 300-byte is allocated on slot 9, which is attacker controlled
Another Case Study

```javascript
var t = {};  
t.__defineSetter__('doc', app.beginPriv);  
t.__defineSetter__('user', app.trustedFunction);  
t.__defineSetter__('settings', function() { throw 1; });  
t.__proto__ = app;  
try {  
  DynamicAnnotStore.call(t, null, f);  
} catch(e) {}  
f();  
function f() {
  app.beginPriv();  
  var file = '/c/notes/passwords.txt';
  var secret = util.stringFromStream(
    util.readFileIntoStream(file, 0)
  );  
  app.alert(secret);  
  app.endPriv();
}
```

CVE-2014-0521 PoC Example
Bypass PlatPal?

An attacker has to *simultaneously* compromise all platforms in order to bypass PlatPal.
Limitations of PlatPal

- User-interaction driven attacks
- Social engineering attacks
  - e.g., fake password prompt
- Other none-determinism to cause divergences
  - e.g., JavaScript \textit{gettime} or RNG functions
Potential Deployment of PlatPal

- Not suitable for on-device analysis.

- Best suited for cloud storage providers which can scan for maldocs among existing files or new uploads.

- Also fits the model of online malware scanning services like VirusTotal.

- As a complementary scheme, PlatPal can be integrated with prior works to provide better prediction accuracy.
Conclusion

• It is feasible to harvest platform diversity for malicious document detection.

• PlatPal raises no false alarms in benign samples and detects a variety of behavioral discrepancies in malicious samples.

• PlatPal is scalable with various ways to deploy and integrate.
Future Works on Diversity Framework

• Implementation diversity
  
  • Case study: PHP interpreters: Zend vs HHVM
Future Works on Diversity Framework

- Implementation diversity

- Case study: PHP interpreters: Zend vs HHVM

- Integration with fuzzing

  - Divergence as an indicator of exception, in addition to crashes and failed assertions
Future Works on Diversity Framework

• Implementation diversity

• Case study: PHP interpreters: Zend vs HHVM

• Integration with fuzzing

  • Divergence as an indicator of exception, in addition to crashes and failed assertions

• Integration with symbolic execution

  • Test whether two functionally similar modules enforce the same sequence and types of checks
Publications

1. Checking Open-Source License Violation and 1-day Security Risk at Large Scale
   Ruian Duan, Ashish Bijlani, Meng Xu, Taesoo Kim, and Wenke Lee
   In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS'17)*

2. PlatPal: Detecting Malicious Documents with Platform Diversity
   Meng Xu, and Taesoo Kim
   In *Proceedings of the 26th USENIX Security Symposium (Security'17)*

   Meng Xu, Kangjie Lu, Taesoo Kim, and Wenke Lee
   In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC'17)*

4. Toward Engineering a Secure Android Ecosystem: A Survey of Existing Techniques
   Meng Xu, Chengyu Song, Yang ji, Ming-Wei Shih, Kangjie Lu, Cong Zheng, Ruian Duan, Yeongjin Jang, Byoungyoung Lee, Chenxiong Qian, Sangho Lee, and Taesoo Kim
   In *ACM Computing Surveys (CSUR) Volume 49, Issue 2, August 2016*

5. UCognito: Private Browsing without Tears
   Meng Xu, Yeongjin Jang, Xinyu Xing, Taesoo Kim, and Wenke Lee.
   In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS'15)*