Towards Automated Integrity Protection of C++ Virtual Function Tables in Binary Programs

Control Flow Based Security Seminar
Matei Pavaluca - WS 2016/2017
Contents

1. Introduction
2. Technical Background
3. High-Level Overview
4. Implementation
5. Evaluation
6. Discussion
7. Conclusion
Purpose

- Automatically protect virtual function tables
- Binary instrumentation
- Replace virtual calls with instrumentation stubs
Contents

1. Introduction
2. Background
3. High-Level Overview
4. Implementation
5. Evaluation
6. Discussion
7. Conclusion
Inheritance and Polymorphism

- Classes contain attributes and members
- Inheriting gives access to all attributes and members
- Declaring virtual functions to change behavior
- Polymorphic classes
- Normal function address: compile time
- Virtual function address: runtime
class B1 {
public:
    void f0() {}
    virtual void f1() {}
    int int_in_b1;
};

class B2 {
public:
    virtual void f2() {}
    int int_in_b2;
};

class D : public B1, public B2 {
public:
    void d() {}
    void f2() {} // override B2::f2()
    int int_in_d;
};

d:
    +0: pointer to virtual method table of D (for B1)
    +4: value of int_in_b1
    +8: pointer to virtual method table of D (for B2)
   +12: value of int_in_b2
   +16: value of int_in_d

Total size: 20 Bytes.

virtual method table of D (for B1):
    +0: B1::f1() // B1::f1() is not overridden

virtual method table of D (for B2):
    +0: D::f2() // B2::f2() is overridden by D::f2()

* source:
https://en.wikipedia.org/wiki/Virtual_method_table
Virtual Function Calls

- Each vFunction has a vTable
- Calling a virtual function means making an indirect call
Calling a Virtual Function

```cpp
class A{
    virtual int Fn(){..;}
};
class B{
    virtual int Fc(){..;}
};
/*single inheritance*/
class C: public B{
    virtual int Fc(){..;}
};
/*multiple inheritance*/
class D: public A, public B {
    ..
};
/* call of overloaded virtual function */
C* p = new C();
p->Fc();
```

1. `mov R, [p]`
2. `add R, offsetFc`
3. `mov R, [R]`
4. `mov this, p`
5. `call R`

R - generic register
1 - dereference
2 - calculate the address of the vTable
3 - dereference
4 - update the “this” pointer
5 - Call into function
Threat Model: Vtable Hijacking

- Exploited by “use-after-free”
- Dangling pointers
- A malicious vTable can be crafted to point to the attacker’s code

Figure 2: The C++ stages, internal low-level operations, and resulting instance’s memory layout of a use-after-free exploitation process utilizing vtable hijacking.
## Usage in the wild

### Table 3: Zero-day attacks using vtable hijacking in-the-wild.

<table>
<thead>
<tr>
<th>CVE</th>
<th>Targeted Application</th>
<th>Module</th>
<th>Vulnerability</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-1690</td>
<td>Fx 17.0.6 (TorBrowser)</td>
<td>xul.dll</td>
<td>use-after-free</td>
</tr>
<tr>
<td>2013-3893</td>
<td>Internet Explorer 9</td>
<td>mshtml.dll</td>
<td>use-after-free</td>
</tr>
<tr>
<td>2013-3897</td>
<td>Internet Explorer 8</td>
<td>mshtml.dll</td>
<td>use-after-free</td>
</tr>
<tr>
<td>2014-0322</td>
<td>Internet Explorer 10</td>
<td>mshtml.dll</td>
<td>use-after-free</td>
</tr>
<tr>
<td>2014-1776</td>
<td>Internet Explorer 8-11</td>
<td>mshtml.dll</td>
<td>use-after-free</td>
</tr>
</tbody>
</table>
1. Introduction
2. Technical Background
3. High-Level Overview
4. Implementation
5. Evaluation
6. Discussion
7. Conclusion
T-VIP Framework

- T-VIP = Towards vTable Integrity Protection
- vExtractor (identifies virtual dispatches) and peBouncer (patches the binary)
- Protect the calls after the payload is injected but before the function is called
Extraction of Virtual Function Dispatches (vExtractor)

- Static instruction slicing and extraction framework
- Generates a control flow graph (CFG)
- Disassembles code to an IL, preserving CFG
- Indirect calls are extracted and set as the criteria for slicing
- Backward Program Slicing to determine virtual calls
Backward slicing

- Tries to find a chain of indirections
- A set of IL instructions is a state
- Starts at an indirect call site and goes backwards
- Stop when a last state is matched or an error occurs
- On success save registers, instruction offsets and addresses
- The instruction which loads the vTable into the register will be protected
Protection of Virtual Function Dispatches (peBouncer)

- Normal calls are replaced by calls to an instrumentation stub that does checks at runtime
- The initial call is copied to the stub to preserve it
- More code can be added based on policies
- A vTable should always reside in non-writable memory
- This is at the base of the instrumentation policies
Policies

- Pnw
  - checks if the memory page where the vTable resides is non writable

- Pnwa
  - includes Pnw
  - checks a random position in the vTable residing above the function to be called
  - the destination is checked for being read only
1. Introduction
2. Technical Background
3. High-Level Overview
4. Implementation
5. Evaluation
6. Discussion
7. Conclusion
vExtractor

- Uses the IDA Pro disassembler
- Converted to REIL, a RISC-like intermediate language
- Easier to parse structure in REIL
- On this intermediate language the amorphous slicing is applied
Amorphous Slicing

- Based on state machines
- Each state contains a few IL instructions
- Indirect call is matched then a state creation is triggered
- The next state is populated with input data from the previous one
- Continues until the start of the indirect call is reached or fails
- When matched, the instrumenter has information about all the offsets, and vTable dereferencing
Figure 3: State design and state transition principle. Wildcard operands are denoted with /.*/
Insertion of Instrumentation Checks

- The instrumentation checks are added in a separate code section
- Replace virtual calls with 32bit relative jumps
- Special measures are taken if there are mismatches in size
Generation of Instrumentation Stubs

● Stubs reside in position-independent sections
● Starts by saving the register environment and ends with restoring it
● Has a generic syntax and replaces needed registers and operands automatically
Virtual Dispatch Instrumentation

- Pnw - 12 instructions
- Pnwa - 23 instructions
- A 64kb lookup bitmap for all usermode pages
- On load the bitmap is populated with the access rights of the pages
- Checking if a vTable is read-only means just checking the page in the bitmap
- The Pnwa is used to protect against ROP attacks
vExtractor Precision

- Based on False-Positives (FP), False-Negatives (FN), True-Positives (TP) and True-Negatives (TN)
- Check against a version instrumented with -fvtable-verify from GCC
- Botan was used as input, because of the extended utilisation of C++ features
vExtractor Precision

- Indirect calls: 6779
- Virtual function dispatches (TP): 6484
- True negatives: 62
- False negatives: 179

- Precision: 0.99
- Recall: 0.97
- F-measure: 0.98
- False positives are due to C structures: st->innerSt->sFn(st, p1, p2)
Runtime of Instrumented Programs - Against GCC

- Using Botan as the input and running Botan’s benchmarks
- Micro benchmarks:
  - Using the GCC: 9,205 median cycle count
  - T-VIP
    - Pnw: 8255
    - Pnwa: 12355
- Macro benchmarks:
  - GCC: 1.0 % median overhead
  - T-VIP: 15.9% median overhead
### Runtime of Instrumented Programs - CPU2006

<table>
<thead>
<tr>
<th>CPU2006</th>
<th>Size</th>
<th>#VD</th>
<th>Native</th>
<th>( P_e )</th>
<th>( P_{nw} )</th>
<th>( P_{nwa} )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Runtime (in s) and overhead (in %)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>rt(s)</td>
<td>rt(s)</td>
<td>ov(%)</td>
</tr>
<tr>
<td>soplex</td>
<td>403K</td>
<td>746</td>
<td>232.25</td>
<td>231.05</td>
<td>-0.52</td>
<td>232.41</td>
</tr>
<tr>
<td>omnetpp</td>
<td>793K</td>
<td>1593</td>
<td>217.12</td>
<td>293.72</td>
<td>35.28</td>
<td>303.48</td>
</tr>
<tr>
<td>povray</td>
<td>1038K</td>
<td>154</td>
<td>164.27</td>
<td>164.22</td>
<td>-0.03</td>
<td>164.36</td>
</tr>
<tr>
<td>dealII</td>
<td>947K</td>
<td>272</td>
<td>360.97</td>
<td>361.75</td>
<td>0.22</td>
<td>363.01</td>
</tr>
<tr>
<td>xalancbmk</td>
<td>3673K</td>
<td>14061</td>
<td>182.97</td>
<td>294.29</td>
<td>60.84</td>
<td>331.98</td>
</tr>
</tbody>
</table>

- **VD** - Virtual dispatches
- **Pe** - empty policy, does no instrumentation, just redirects every call
Runtime of Instrumented Programs - Browsers

- Instrumented certain modules of Firefox on IE on different versions
- Sun-Spider and Kraken were used as benchmarks

<table>
<thead>
<tr>
<th>App.</th>
<th>Module</th>
<th>#IC</th>
<th>#Slices</th>
<th>#Filtered</th>
<th>#Instr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fx 17.0.6</td>
<td>xul.dll</td>
<td>66,120</td>
<td>53,268</td>
<td>73</td>
<td>53,195</td>
</tr>
<tr>
<td>IE 8</td>
<td>mshtml.dll</td>
<td>23,682</td>
<td>19,721</td>
<td>3,117</td>
<td>16,604</td>
</tr>
<tr>
<td>IE 9</td>
<td>mshtml.dll</td>
<td>64,721</td>
<td>53,312</td>
<td>7,735</td>
<td>45,577</td>
</tr>
<tr>
<td>IE 10</td>
<td>mshtml.dll</td>
<td>56,149</td>
<td>44,383</td>
<td>5,515</td>
<td>38,868</td>
</tr>
</tbody>
</table>
Runtime of Instrumented Programs - Browsers

- Overall degradation:
  - Pe: 2.1%
  - Pnw: 1.6%
  - Pnwa: 2.2%
Efficiency

- Fixes vulnerabilities:
  - CVE-2013-3897
  - CVE-2013-3893
  - CVE-2013-1690
  - CVE-2013-2556
  - CVE-2014-0322
1. Introduction
2. Technical Background
3. High-Level Overview
4. Implementation
5. Evaluation
6. Discussion
7. Conclusion
Limitations

- Misses 2.6% of virtual dispatches
- Compilation problems when testing
- There are ways to circumvent the protection
- Poor stealthiness, can be exploited
Contents

1. Introduction
2. Technical Background
3. High-Level Overview
4. Implementation
5. Evaluation
6. Discussion
7. Conclusion
Conclusion

- The method works against known vTable exploitation attacks
- Has a fairly good precision and performance on web browsers (2.2% overhead)
- Poorer performance in protecting other types of code
- Instruments binaries, versatile
Questions?
Literature