T-VIP
Towards automated integrity protection of C++ virtual function tables in binary programs

Seminar:
Code Reuse Attacks and Defenses against them

Michael Edelmann
Technische Universität München
Fakultät für Informatik

03.07.2017
Outline

1) How virtual tables work
2) Attacking virtual tables
3) Counter measures: The T-VIP approach
4) Conclusions
About C++ classes/objects

- C++ objects mainly containers for attributes
- Normal methods get object as first parameter (thiscall calling convention)
- Derived classes can override methods from their parents (virtual functions)
- Type of object not always clear at compile time
How virtual tables work

Writable memory section (r w -)

Base-object memory layout

<table>
<thead>
<tr>
<th>Address</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x...00</td>
<td>vTable-ptr</td>
</tr>
<tr>
<td>0x...08</td>
<td>First member</td>
</tr>
</tbody>
</table>

Child-object memory layout

<table>
<thead>
<tr>
<th>Address</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x...40</td>
<td>vTable-ptr</td>
</tr>
<tr>
<td>0x...48</td>
<td>First member</td>
</tr>
</tbody>
</table>

Readonly memory section (r - -)

vTable for Base

<table>
<thead>
<tr>
<th>Address</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x...00</td>
<td>addr of Base::vFunc1</td>
</tr>
<tr>
<td>0x...08</td>
<td>addr of Base::vFunc2</td>
</tr>
</tbody>
</table>

vTable for Child

<table>
<thead>
<tr>
<th>Address</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x...00</td>
<td>addr of Child::vFunc1</td>
</tr>
<tr>
<td>0x...08</td>
<td>addr of Child::vFunc2</td>
</tr>
</tbody>
</table>

// Given: Object's address in rbx
mov rax, [rbx]    // vTable address into register
mov rdi, rbx      // setup this-pointer
call qword [rax + 8*i] // call i-th virtual function
Attacking virtual tables

Heap (r w -)

Object memory layout

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x...00</td>
<td>vTable-ptr</td>
</tr>
<tr>
<td>0x...08</td>
<td>First member</td>
</tr>
</tbody>
</table>

Stack / Heap (r w -)

fake vTable for Class

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x...00</td>
<td>addr of injected code</td>
</tr>
<tr>
<td>0x...08</td>
<td>addr of injected code</td>
</tr>
</tbody>
</table>

Readonly memory section (r - -)

vTable for Class

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x...00</td>
<td>addr of Class' vFunc1</td>
</tr>
<tr>
<td>0x...08</td>
<td>addr of Class' vFunc2</td>
</tr>
</tbody>
</table>

Code section (r - x)

Some type of useful gadget
Attacking virtual tables

- Use after free
  ```cpp
  SomeObject *o = new SomeObject();
  // ...
  delete o;
  // Write some data to heap
  o->vFunc();
  ```

- vTable address could be overwritten

- Call to vFunc can result in execution of user-selected code
Counter measure: T-VIP

• Problem
  − Trying to call a function referenced from writable memory (objects vTable pointer)

• Assumptions
  − No source code
  − No debugging symbols or runtime type information

• Solution
  − Identify calls to virtual functions (vExtractor)
  − Precede calls with sanity checks (PeBouncer)
Identify virtual function calls

- Search for indirect call instructions
- Abstract assembly to RISC IL code
  - Reduce dependency on specific instruction set
  - Reduce number of possible code representations
- Search backwards to resolve indirect call
- Decide if we actually found a virtual call
Machine code abstraction
REIL: Reverse Engineering Intermediate Language

• Reduce instruction sets to common elements
• 17 instructions (6 arithmetic, 3 logic, 3 data transfer, 2 conditional (BISZ, JCC), 3 other)
• Simple syntax
  - instr src1, src2, dst
  - instr (src1, type1), (src2, type2), (dst, type3)
Backward search

• Use state-machines
• States are sets of instruction patterns
• State change if instruction pattern is matched
  - e.g. matching mnemonic & dst-register
  - Copy src of old state to dst in new state
• Assume virtual call, if end-state is reached
State machines

Dereference Detection

Dereference 0 → 1 → 2 → 4 → 8
// Object into rbx
mov rax, [rbx]
mov rdi, rbx
call qword [rax]

thiscall Detection
Backward search

• Problem: Some constructs like nested C-structs look like virtual calls
  - `outerSt->innerSt->fn(outerSt, p1, p2)`

• Solution: Profiling Phase
  - Generate executable
  - Execute & try to visit all virtual call sites
  - Assumed vTable in writable memory → No virtual call
Change Executable

- Replace some instructions with relative jump to checking code (instrumentation stub)
- Insert instrumentation stub
  - Original instructions
  - Integrity Policy
  - Jump to the address after original instruction

```plaintext
[...] 0x7167d54d: push ebx 0x7167d54e: mov edi, 0x80004002 0x7167d553: call dword [eax]
[...] 0x7167d54d: push ebx 0x7167d54e: jmp STUB_7167d54e 0x7167d553: call dword [eax]
[...] STUB_7167d54e: mov edi, 0x80004002 [...] ## INSTRUMENTATION jmp 0x7167d553
```
Inserting jump

- If original opcode size ≥ jump opcode size
  - Replace and fill remaining bytes with NOPs
- Else
  - Continue replacing subsequent instructions
- Problematic instructions
  - e.g. relocations, basic block leaders/terminators
  - On encounter: Stop. Search upwards instead
  - If enclosed between two problematic instructions
    - Overwrite one with illegal instruction
    - Install vectored exception handler as trampoline
Included integrity policies
nw (12 assembly instructions)

• Modify executable
  – Create 64KB lookup-table

• During runtime
  – Set bit for non-writable memory pages
  – Transform vTable address with simple operations
    \((addr \& 0xffffffffc000) \gg 0xc\)
  – Query lookup table to determine memory protection

• Can be circumvented by finding non-writable address which points to gadget
Included integrity policies
nwa (23 assembly instructions)

- Choose additional entry above virtual function call at random
- Dereference entry
- Query lookup table
  - Should point into code section $\rightarrow$ non-writable
Writing own integrity policies

- Annotation feature to modify instrumentation code
  - after assembly
  - before insertion into new code section
- Assembly contains <keywords>
  - Replaced by x86-mnemonic & hash of keyword
- Before insertion as a stub
  - Search for hashes
  - Replace with adjusted opcodes (specific for keyword & instrumentation stub)
Utilizing OS APIs

- Service libraries
  - Exported functions which wrap API functions
- Modify program entry point to load service library
- Allows for full support of ASLR & NX

<table>
<thead>
<tr>
<th>New data section (Custom IAT)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x...800</td>
<td>[Address to lib func1]</td>
</tr>
<tr>
<td>0x...808</td>
<td>[Address to lib func2]</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Service Library instrumentation stub

```assembly
### INSTRUMENTATION START ###
## Prolog
pushad
pushfd
call $+5
sub dword [esp], 7  // calculate stub IP
##

## annotation: move register with vtable to ecx
<MOV_ECX_VTABLE_REG>

mov ebp, dword [esp]  // get IP
push ecx  // push vtable address

## annotation: service lib function (outputs vtable)
call dword [ebp + <IAT.vProtSrv.debugOut>]

add esp, 4  // remove vtable address

## Epilogue
add esp, 4
popfd
popad
##

### INSTRUMENTATION END ###
```

- Save GPRs to stack
- Save flags to stack
- Save IP to stack
- Indirect call with register & annotation keyword
  - keyword contains library & function name
  - PeBouncer replaces keywords
Conclusions

- vExtractor precision: 97.4%
- Botan-Benchmarks (90 highly-demanding cryptographic algorithms)
  - Median Overhead: 15.9% (37 smaller than 2%)
    - GCC approach: 1.0% (47 smaller than 2%)
    - Reason: Vectored Exception handlers
- Kraken & SunSpider benchmark with Firefox
  - Average overhead: 1.8% nw, 2.1% nwa
- In most cases „nw“ policy is enough
References

- *Towards Automated Integrity Protection of C++ Virtual Function Tables in Binary Programs*, Robert Gawlik and Thorsten Holz

- *[TR-HGI-2014-004]*, Robert Gawlik and Thorsten Holz

- *REIL: A platform-independent intermediate representation of disassembled code for static code analysis*, Thomas Dullien and Sebastian Porst