one aspect of
Security on JIT VMs
and more
Richie++ / richie / gera@corest.com
Security?

• Does your application need security?
• Do you do anything for its security?
one aspect of
Security on JIT VMs
and more
Scenario

- VM installed on user's machine (your application)
- Attacker provides content (behavior)
- Attacker escapes VM restrictions (if any)
- Attacker accesses private information
• Smalltalk stack == native stack
• Instance variables are accessed directly
• Contexts are stored in native stack
nativizing VMs
Cincom VW

test: arg

^self with: 0 with: inst with: arg

- Smalltalk stack \(\approx\) native stack
  - First arguments go in registers
  - Instance variables are accessed “directly” (some)
  - Contexts are stored in native stack
Conding in assembly
stack operations

- **Push**: add arbitrary things to the stack
- **StoreTemporary**: random access to negative offsets
- **PopR, DropTos**: arbitrarily move the stack pointer
- **NoFrameProlog**: skips saving/restoring the FP and SP
Escaping Digitalk VM

- Drop top of stack
- Overwrite return address with argument
- Return

@1: 7CBC6D: push EBP
    7CBC6E: mov EBP, ESP
    7CBC70: push EAX
    7CBC71: mov ESI, EAX
    ...

@2: 7CBC82: add ESP, 10
    7CBC85: push [EBP+4]
    7CBC88: mov ESP, EBP
    7CBC8A: pop EBP
    7CBC8B: mov ESI, [EBP-4]
    7CBC8E: ret NEAR

; 1<05> DropTosN 4
; 4<55> PushArgument1
; 5<48> Return
Unbalancing the stack

- Caller pushes no arguments
- Callee cleans 5 arguments, unbalances stack
- Caller can modify *protected* values (return address)

```
@1: 7F2CDD: push EBP
    7F2CDE: mov EBP, ESP
    7F2CE0: push EAX
    7F2CE1: mov ESI, EAX
    7F2CE3: push 100BCF14
    7F2CE8: cmp ESP, [10028CD4]
    7F2CEE: inc EBX
    7F2CEF: jbe @5
    7F2CF1: inc EBX

@2: 7F2CF2: mov ESP, EBP ; 1<48> Return
    7F2CF4: pop EBP
    7F2CF5: mov ESI, [EBP-4]
    7F2CF8: ret 14 NEAR
```
Escaping Cincom VM

**Selector:** #test:
**Arguments:** 8
**Frame size:** 10

- Caller pushes 1 argument
- Callee cleans 7 arguments, unbalances stack
- Caller can modify *protected* values (return address)
• Attacker transfers a CompiledMethod and activates it
• Attacker escapes the VM, and accesses the OS
• OS does not provide Application storage isolation
• Attacker gets stored passwords and credit cards
Securing Smalltalk
(some ideas)

fileMeta := 0 class class class allInstances
detect: [:metaClass |
    metaClass instanceClass name = 'File'].
fileMeta instanceClass openReadOnly: '...\savedPasswords'

File pathNameReadOnly: 'temporaryFile.dat'

File pathNameReadOnly: '...\savedPasswords'

PushR, PushR, DropTos2, Return
PushR, PushR, Return

• Reachability
• Sandboxing
• Verifier

• Are all three necessary?
Escaping Digitalk VM

SmallInteger >> #readMemory
LoadInstance1
Return

@2: 796DA2: mov EAX, [ESI]
796DA4: mov ESP, EBP
796DA6: pop EBP
796DA7: mov ESI, [EBP-4]
796DAA: ret 4 NEAR

; 1 <7F> LoadInstance1
; 2 <48> Return

SmallInteger >> #writeMemory:
LoadArgument1
StoreInstance1
Return

@2: 7FCA82: mov EAX, [EBP+8]
7FCA85: mov [ESI], EAX
7FCA87: call 1001AEA0
7FCA8C: mov ESP, EBP
7FCA8E: pop EBP
7FCA8F: mov ESI, [EBP-4]
7FCA92: ret 4 NEAR

; 1 <50> LoadArgument1
; 2 <96> StoreInstance1
; 3 <48> Return

- Arbitrary memory read
- Arbitrary memory write
Escaping Cincom VM

SmallInteger >> #readMemory:
OpLoadInst
OpReturn

8EA1F09: mov EDX, [ESI] ; OpStorePopInst
8EA1F0B: mov EBX, [EDX]
...
8EA1F20: leave
8EA1F21: ret NEAR ; OpReturn

SmallInteger >> #writeMemory:
OpLoadTemp
OpStorePopInst
OpReturn

8EA1F09: mov EDX, [ESI] ; OpStorePopInst
8EA1F0B: mov [EDX], EBX
...
8EA1F20: leave
8EA1F21: ret 4 NEAR ; OpReturn

- Arbitrary memory read
  (#[0 0 0 16r78 16r56 16r34 16r12] copyToHeap asInteger / 4) readMemory
- Arbitrary memory write
[Audience hasQuestions] whileTrue: [
  self answer: Audience nextQuestion]
Further understanding
better assessment
Documenting Bytecodes

BytecodeNativizerPushR
assembler pushR

Assembler >> #pushR
self assembleByte: 16r50 " push eax "

BytecodeNativizerDropTos1
assembler dropTos: 1

Assembler >> #dropTos: index
self " sub esp, index * 4 "
assemble: #[16r83 16rC4];
assembleByte: index * 4

BytecodeNativizerDropTosN
| idx |
idx := self nextIndex.
assembler dropTos: idx

BytecodeNativizerLoadInstN
assembler loadFromInstance: self nextIndex

Assembler >> #loadFromInstance: index
" We need mov eax, [esi + index*4] "
self assembleByte: 16r8B.
index = 1 ifTrue: [
  ^self reg: 0 mod: 0 rm: 6].
index abs > 31
  ifTrue: [
    self assembleLong: index - 1 * 4]
  ifFalse: [...]
Testing Bytecodes

Templates

loadInstance1

```
1   <7F> LoadInstance1
2   <48> Return
```

| a |
^testSelector

loadInstanceNoProlog1

```
1   <02> NoFrameProlog
2   <7F> LoadInstance1
3   <48> Return
```

^testSelector

Test

testSameAsOriginal: cm
| original documentation |
original := CompiledMethodNativizer
originalNativize: cm.
documentation := CompiledMethodNativizer
nativize: cm.
self assert: original == documentation

shortForwardTestJumpFalse

```
1   <0E> LoadTrue
2   <1B> TestJumpFalse 6
5   <14> LoadSmallInteger1
6   <49> ReturnSelf
```

true ifTrue: [1].
So far so good... but does the generated code work?
Two worlds unite

PushSmallInteger 1234
PushArgument1
LoadSelf
SendSelector1
Return

methodLookup() → rtCompile()
Two worlds unite

PushSmallInteger 1234
PushArgument1
LoadSelf
SendSelector1
Return

methodLookup() → rtCompile()

rtCompile: aCompiledMethod
Transcript
  show: 'got callback ';
  Show: aCompiledMethod printString; cr
cr.

aCompiledMethod selector = #testMethod ifTrue: [
  ^self nativize: aCompiledMethod].

^0
What's next?

- Debugging JIT
- Frozen code
- All in Smalltalk
[Audience hasQuestions] whileTrue: [
    self answer: Audience nextQuestion].

Audience do: [:you | self thank: you].

self returnTo: Audience
• We understand why it's important that all this VMs have security: We use most of them every day in some way or another, and through them we extend our trust to untrusted mobile code applications from we download from unknown sources.
• They have all gone, directly or indirectly, through some security audits and, one way or another, their developers today care about security issues.
• Smalltalk has grown as the very open "socialist" environment we all love, were we just trust everybody. The VMs developers community has not really payed much attention to security (not at least from a mobile code perspective)
• Time has come for mobile code to also reach Smalltalk (browser plugins, Croquet objects with their own behavior, Scratch/EToys projects, seaside hosting)
Security?

- Does your application need security?
- Do you do anything for its security?

- Any application installed on a computer can open the door to an attacker, in our scenario we are assuming mobile code (maybe embedded in other type of content)
- The VM is supposed to provide some sandboxing, if the sandboxing can be broken the trust chain can be abused:
  - the user trusts the VM
  - the VM trusts/verifies the mobile code
  - the mobile code fools the VM
• We understand why it's important that all this VMs have security: We use most of them every day in some way or another, and through them we extend our trust to untrusted mobile code applications from we download from unknown sources.

• They have all gone, directly or indirectly, through some security audits and, one way or another, their developers today care about security issues.

• Smalltalk has grown as the very open "socialist" environment we all love, were we just trust everybody. The VMs developers community has not really payed much attention to security (not at least from a mobile code perspective)

• Time has come for mobile code to also reach Smalltalk (browser plugins, Croquet objects with their own behavior, Scratch/EToys projects, seaside hosting)
Scenario

- VM installed on user's machine (your application)
- Attacker provides content (behavior)
- Attacker escapes VM restrictions (if any)
- Attacker accesses private information

- Any application installed on a computer can open the door to an attacker, in our scenario we are assuming mobile code (maybe embedded in other type of content)
- The VM is supposed to provide some sandboxing, if the sandboxing can be broken the trust chain can be abused:
  - the user trusts the VM
  - the VM trusts/verifies the mobile code
  - the mobile code fools the VM
- Smalltalk stack == native stack
- Instance variables are accessed directly
- Contexts are stored in native stack

- Smalltalk is compiled to Bytecode
- Bytecode is nativized
- Smalltalk is directly compiled to Assembly
- Here we can see
  - Smalltalk stack is kept in the native stack (push nativized to push)
  - Contexts are normal native stack frames
    - return addresses managed through native call/ret
  - Arguments are accessed through the native frame pointer
  - Local variables are accessed through the native frame pointer
  - The receiver is saved in a local
  - Instance variables are accessed with direct memory accesses without any checks
- If we could arbitrarily access any indexed argument we could corrupt the return address or saved receiver (from other frames)
nativizing VMs
Cincom VW

• Smalltalk stack =~= native stack
  • First arguments go in registers
  • Instance variables are accessed “directly” (some)
  • Contexts are stored in native stack

• Here we can see
  • Smalltalk stack is kept in the native stack (push nativized to push)
    • Contexts are normal native stack frames
      • return addresses managed through native call/ret
    • after 3 args they accessed through the native frame pointer
    • Local variables are accessed through the native frame pointer
    • The receiver is saved in a local
    • Some instance variables are accessed with direct memory accesses without any checks, some are accessed through a routing that MAY do checks.
  • If we could arbitrarily write any indexed argument we could corrupt the return address, saved receiver or locals (from other frames)
  • If we could arbitrarily manipulate any indexed instance variable we could corrupt the object space.
## Disassembler

### Method Source

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Comment</th>
<th>Selector</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>754AE0</td>
<td>push EBX</td>
<td>01000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>754A9F</td>
<td>cmp ESP, [1000BCD4]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>754A9E</td>
<td>inc EBX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>754A9D</td>
<td>jbe $5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>754A9C</td>
<td>inc EBX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>754A9B</td>
<td>push ESP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>754A9A</td>
<td>mov ESP, [EBP+4]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>754A99</td>
<td>pop ESP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>754A98</td>
<td>mov EAX, [10005476]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>754A97</td>
<td>call 754B40</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>754A96</td>
<td>mov ESP, EBP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>754A95</td>
<td>ret NEAR</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Code Attributes
- **Do It**: Display method
- **Debug It**: Browse in Target Process
- **Browse It**: WindowsBuilder
- **Local**: Debugger
- **Locations**: Local variables
- **View**: Local implementations
- **Send**: Project senders
- **Receive**: Project implementors
- **Display**: Matching selectors
- **Find**: Find symbol
- **Save**: Undo
- **Format**: Show byte codes

---

Disassembler output demonstrating the disassembly of a method in a program, showing the disassembly of instructions at specific memory locations.
• **Push**: add arbitrary things to the stack
• **StoreTemporary**: random access to negative offsets
• **PopR, DropTos**: arbitrarily move the stack pointer
• **NoFrameProlog**: skips saving/restoring the FP and SP
Escaping Digitalk VM

- Drop top of stack
- Overwrite return address with argument
- Return

@1: 7CBC6D: push EBP
     7CBC6E: mov EBP, ESP
     7CBC70: push EAX
     7CBC71: mov ESI, EAX
     ...

@2: 7CBC82: add ESP, 10
     7CBC85: push [EBP+4]
     7CBC86: mov ESP, EBP
     7CBC88: pop EBP
     7CBC89: mov ESI, [EBP-4]
     7CBC8E: ret NEAR
Unbalancing the stack

1. Caller pushes no arguments
2. Callee cleans 5 arguments, unbalances stack
3. Caller can modify *protected* values (return address)
Escaping Cincom VM

- Caller pushes 1 argument
- Callee cleans 7 arguments, unbalances stack
- Caller can modify *protected* values (return address)
• Attacker transfers a CompiledMethod and activates it
• Attacker escapes the VM, and accesses the OS
• OS does not provide Application storage isolation
• Attacker gets stored passwords and credit cards
Securing Smalltalk
(some ideas)

```smalltalk
fileMeta := 0 class class class allInstances
detect: [:metaClass |
    metaClass instanceClass name = 'File'
].
fileMeta instanceClass openReadOnly: '\...\savedPasswords'

File pathNameReadOnly: 'temporaryFile.dat'
File pathNameReadOnly: '\...\savedPasswords'

PushR, PushR, DropTos2, Return
PushR, PushR, Return
```

- **Reachability**
- **Sandboxing**
- **Verifier**

- Are all three necessary?

- In a fully dynamic and reflective system reachability is hard to constrain

- Sandboxing, ACL or permissions must be implemented in the VM itself. They could be bypassed, most likely, if implemented in the image.

- A bytecode verifier is mandatory in any kind of nativizing VM, and strict checks are also required in an interpreter.
Escaping Digitalk VM

- Arbitrary memory read
- Arbitrary memory write
Escaping Cincom VM

- Arbitrary memory read
  (#[0 0 0 16r78 16r56 16r34 16r12] copyToHeap asInteger / 4) readMemory

- Arbitrary memory write
[Audience hasQuestions] whileTrue: [
  self answer: Audience nextQuestion]
• Just randomly looking how each bytecode is nativized is not enough to assess the security of the system, we need to understand all possible variations for each bytecode.

• So we need to understand and document the workings of the JIT nativizer, and what better documentation that something which can be debugged. So, as we all learned long time ago with the original specification of the Smalltalk VM, we started documenting not in .doc, but in .st
Documenting Bytecodes

BytecodeNativizerPushR
assembler pushR

Assembler >> #pushR
self assembleByte: 16r50 /* push eax */

BytecodeNativizerDropTos1
assembler dropTos: 1

Assembler >> #dropTos: index
self /* sub esp, index * 4 */
assemble: #16r83 16rC4;
assembleByte: index * 4

BytecodeNativizerDropTosN
| idx |
idx := self nextIndex.
assembler dropTos: idx

Assembler >> #loadFromInstance: index
" We need mov eax, [esi + index*4]"
self assembleByte: 16r8B.
index = 1 ifTrue: ['self reg: 0 mod: 0 rm: 6].
index abs > 31
ifTrue: [
  self assembleLong: index - 1 * 4]
ifFalse: [...]
Testing Bytecodes

**Templates**

```
loadInstance1
1  <7F>  LoadInstance1
2  <48>  Return
   | a |
   ^testSelector
```

```
loadInstanceNoProlog1
1  <02>  NoFrameProlog
2  <7F>  LoadInstance1
3  <48>  Return
   ^testSelector
```

**Test**

```
testSameAsOriginal: cm
   | original documentation |
   original := CompiledMethodNativizer
   originalNativize: cm.
   documentation := CompiledMethodNativizer
                    nativize: cm.
   self assert: original == documentation
```

```
shortForwardTestJumpFalse
```

```
true ifTrue: [1].
```
So far so good… but does the generated code work?
Two worlds unite

PushSmallInteger 1234
PushArgument1
LoadSelf
SendSelector1
Return

methodLookup()

rtCompile()
Two worlds unite

```
PushSmallInteger 1234
PushArgument1
LoadSelf
SendSelector1
Return

rtCompile: aCompiledMethod
Transcript
  show: 'got callback ';
  Show: aCompiledMethod printString; cr
  cr.

aCompiledMethod selector = #testMethod ifTrue: [
  *self nativize: aCompiledMethod].

^0
```
What's next?

- Debugging JIT
- Frozen code
- All in Smalltalk
[Audience hasQuestions] whileTrue: [self answer: Audience nextQuestion].

Audience do: [:you | self thank: you].

self returnTo: Audience