Automated unpacking

Malware Analysis Seminar
Meeting 5
Cody Cutler, Anton Burtsev
Code packing
Types of packers

- Traditional
- Shifting decode frame
- Code virtualization
Taxonomy of polymorphism
Syntactic changes

- Change of code's syntactic structure
  - Semantics remains unchanged
  - Borrows techniques of code obfuscation
- Evade signature based detection of traditional antivirus software
• Dead code insertion
  
push %ebx
pop %ebx

• Instruction substitution
  
mov $0, %eax -> xor %eax, %eax

• Variable renaming & register reassignment
  
mov $0, %eax             mov $0, %ebx
mov $1, %ebx             mov $1, %ecx
add %eax, %ebx           add %ebx, %ecx
push %ebx                push %ecx
call $0x80403020         call $0x80403020

• Code reordering
  
  • Change syntactic order of the code
  
  • Semantic execution path remains unchanged
- Branch obfuscation
  - Hide the target of a branch
  - Structured Exception Handling
  - Indirect branching
    ```assembly
mov   $0x80402030, %eax
jump  *%eax
```
- Branch functions
- Branch inversion
  ```assembly
jc  0x80403020 -> cmc      #complement curry flag
    jnc $0x804030
```
- Branch flipping
  ```assembly
jz  0x80403020 ->     jnz L
    jmp $0x804030
L:
```
• Opaque predicate insertion
  • Always evaluates to the same result
  • However it's hard to know this result statically
    - Used for both control flow, and data values
      
      ```
      mov $1, %eax
      jnz $0x80403020
      ```
Automated unpacking: detecting packed code
Detection

- Signature-based detection
  - PEiD
- Entropy analysis [Bintropy]
  - Statistical measure of the amount of information in a block of data
  - Packed and encrypted code has high entropy
- Limitations
  - Packers can lower the entropy intentionally
  - Entropy analysis can miss simple obfuscation
Detection

• Behavior based
  • Monitor execution
  • Detect if previously modified memory is executed
• Limitations
  – Can't distinguish self-modified and packed code
Program feature classification

- **Program features**
  - Number of standard and non-standard sections
  - Number of executable sections
  - Number of readable/writeable/executable sections
  - Number of entries in the import table
- **Some static program features remain invariant**
  - Byte and instruction level features perform poorly
  - But don't require undecidable disassembly
  - Code normalization might help
    - But it's not sound
Automated unpacking: static approaches
Code normalization

- The goal is to undo obfuscation
- Code reordering
  - Reliable for unconditional jumps
  - “In a normalized CFG, each CFG node with at least one unconditional-jump immediate predecessor also has exactly one incoming fall-through edge”
- Semantic nops
  - Abstract interpretation
Control flow and call graphs

- More invariant
  - Fail to reconstruct precise CFG in face of...
- Opaque predicates (misleading branch targets)
  - Detect opaque predicates
    - Remove them with abstract interpretation
- Pointers and indirection
- Some models ignore indirect branches all together
  - Accept a less accurate representation
- Alias analysis (Value-Set analysis)
  - Tries to detect all possible values for the pointers
Feature classification

- Data-flow and dependence analysis
  - Hard in the presence of pointers
- API calls
  - Fail in face of stolen bytes which obscure API calls
Automated unpacking: dynamic approaches
PolyUnpack

- Generate static code view
- Identify generated instructions
  - Compare at run-time if instruction is in the static view, if not, it was dynamically generated
// Step 1: Static Analysis

// Disassemble P to identify code and data. Partition blocks of code separated by non-instruction data into sequences of instructions i0, ..., in. These sequences form the set I (the static code view). I will be repeatedly queried in the dynamic analysis step to detect if P is executing unpacked code.

// Step 2: Dynamic Analysis

// Execute P one instruction at a time. Pause execution after each instruction and acquire the current instruction sequence by performing in-memory disassembly starting at the current value of the pc until non-instruction data is found. Compare the current instruction sequence with each instruction sequence in the set I. If the current instruction sequence is not a subsequence of any member of I, then it did not exist in the static code view of P (i.e., it is unpacked code being executed).
PolyUnpack: implementation

- Command-line windows tool
  - Software and hardware breakpoints to implement single-stepping
  - [www.ollydbg.de/srcdescr.htm](http://www.ollydbg.de/srcdescr.htm) library for disassembling
  - OllyDump for dumping
- Careful handling of DLL code
  - Also linked dynamically
Renovo

• Part of BitBlaze
  • Implemented on top of TEMU, extension of QEMU
• Shadow memory
  • Tracks clean (unmodified), and dirty (modified) memory
  • After a block in a dirty memory is executed, Renovo dumps dirty memory, and marks it as clean again
• Tracks processes with CR3
Saffron

- Same idea but uses binary instrumentation to control the program
  - Pin
- Later implementation relies on the Windows page-fault handler modification
  - Tracks memory modifications
Criticism

- Simplistic models
- Heavyweight
- A typical AV solution uses a combination of
  - x86 emulator
  - application level OS emulation
Automated unpacking: dealing with code virtualization
Code virtualization

• Themida
  • Translates x86 code into another language
    – RISC-64, RISC-128, CISC, CISC-2
  • Randomizes instruction encoding
  • Interprets new language

• VMProtect
  • Stack based RISC
Static approach

- Compiler front-end which takes a v-code language
- Recompile in x86
- Observations
  - v-code language is derived from a family of templates
  - High similarity
People do that

- Reverse engineer the VM
  - With the help of dynamic tools
- Implement a disassembler
  - IDA Pro plugin 5K LOC of C++
- Disassemble byte code and convert into IR
- Apply compiler optimizations
- Generate x86 code
Rotalume

- QEMU based dynamic analyzer
  - Record a trace of execution
  - Identify the virtual program counter (VPC)
    - Abstract variable binding
    - Associate each memory fetch with an index variable
    - Deal with x86
  - Identify v-code regions
  - Identify syntax and semantics of v-code operations
    - CFG and taint analysis
Acknowledgements

- Survey of Unpacking Malware. Silvio Cesare.
- Unpacking virtualization obfuscators. Rolf Rolles. In WOOT'09.