CodeXt: Automatic Extraction of Obfuscated Attack Code from Memory Dump

Ryan Farley and Xinyuan Wang
George Mason University
Information Security, the 17th International Conference
ISC 2014, Hong Kong
Problem

• Need to automate upon detection in memory
  – Avoid substantial manual effort
    • Automatically recover malcode
    • Extract/unpack/recover attack code
  – Memory dump, transient artifacts
Existing Tools

• Only work with known boundaries
  – Typically designed for full binaries
    • e.g., PE files
  – Things get nasty without given boundaries
    • Or are arbitrary byte streams

• Don’t generically handle
  – Malformed, Misaligned
  – Obfuscated, Armored
Solution: CodeXt

- Discovers executable code within memory dump
  - Upon realtime detection
- Extracts packed or obfuscated malcode
  - First to generically handle Incremental and Shikata-ga-nai
Solution: CodeXt

- Framework built upon S2E
  - Selective means QEMU vs KLEE (LLVM)
  - Decision made per basic block
CodeXt Output

• Instruction Trace of executed instructions
  – Grouping of fragments into chunks
  – Reveals original and unpacked malcode
  – Assisted by a translation trace

• Data Trace of memory writes
  – Intelligent memory update clustering
  – Multi-layer snapshots

• Call Trace of system calls
  – With CPU context
Problems + Challenges + Solutions
Handling Byte Streams

• S2E expects well structured binaries
  – We wrap the binary for execution

• S2E uses basic block granularity
  – Our modified QEMU translation returns more info
  – We leverage translation and execution hooks to verify
Recognizing Code

• Avoiding the Halting Problem
  – No infinite loops
  – Caps on executed instructions
    • Different types: target, non-target, system

• False cognates
  – Illegal first instructions
  – False jumps into suffix

• Many substrings
  – Matched code fragment: ends on system call, EAX within range
Dealing with Code Fragments

- **Fragmentation**
  - Clustering into Chunks, adjacency, execution trace

- **Density**
  - Usage: Executed/Range
  - Overlay: Unique executed/Range over snapshots

- **Enclosure**
  - Continuous executable bytes adjacent to end

\[ \text{S2E} \left( \text{offset}_i, \text{offset}_n \right) \rightarrow \text{Fragments} \rightarrow \text{Match} \]
Defeating Obfuscation

• FPU instructions, fnstenv
  – Added small change to QEMU to comply

• Intra-basic block self-modification
  – We know address range of each translated block
  – During execution we track writes
  – If any write is to same block we retranslate block

• Emulator detection
  – Tested for a set of obscure instructions used as canaries
Results
Experiments

• Hidden code search
  – 1KB to 100KB buffers, 40B to 80B shellcodes
  – Filled with either null, live-capture, or random bytes
  – Varied assistance data: EIP, EAX, both, neither

• Accuracy
  – De-obfuscation, Anti-emulation detection
  – Various packers mentioned in previous research
  – In-shop: Junk code insertion, Ranged xor, Incremental

• Symbolic Branching
Multi-Layered Encoders

\[
xor(key2, 5, 10, xor(key1, 30, 10, xor(key1, 10, 10)))
\]

deh-junk()
Incremental Encoder

Original memory

First snapshot

Second snapshot

Third snapshot

Decoder3 w/ K3

Encoded code, data

Transient code 1

Decoder2 w/ K2

Layer 3 decoded

Transient code 2

Layer 2 decoded

Decoder1 w/ K1

Layer 1 decoded

Layer 3 decoded

Layer 3 decoded

Layer 2 decoded

Layer 2 decoded

Layer 1 decoded

First snapshot

Second snapshot

Third snapshot
Symbolic Conditionals

```plaintext
...  
y=0; z=1;  
if (x>=10)  
y=func3();  
else if (x>=0)  
y=func1();  
if (y==0)  
z=0;  
if (y==1 && z==0)  
z=4;  
...  
```
Conclusion

• Emulation is heavy-weight, but
  – Accurate and enables anti-anti-sandbox techniques
  – OS independent
• Symbolic analysis engine opens avenues
  – Taint propagation and analysis
  – Fuller branch exploration and pruning heuristics
• CodeXt
  – Accurately pinpoints and models even highly obfuscated code in adverse conditions.
Current/Future Development

• Full binary (ELF/PE) support
  – Modeling unmodified executables
  – Without source code

• Data and code based taint analysis
  – Can mark any input
    • e.g., all network socket reads
  – Follow not only by data, but instruction influence
Thank you for your time
• Any questions?

Post-talk, please feel free to contact us
• ryanfarley@ieee.org
• xwangc@gmu.edu