SD-Dyninst

Static-Dynamic Malware Analysis

Dyninst

UNIVERSITY OF MARYLAND 18 56

WISCONSIN UNIVERSITY OF WISCONSIN–MADISON

The Malware Binary

Why is analyzing malware hard?

- Packed code: the binary's malicious code is not generated until runtime
- Self-modifying code: overwrites can change the code's behavior
- Obfuscated code: what code *is* statically visible is hard to analyze
- *These techniques are pervasive!* 90% of malware is analysis-resistant

Before Execution

Unpacking loop

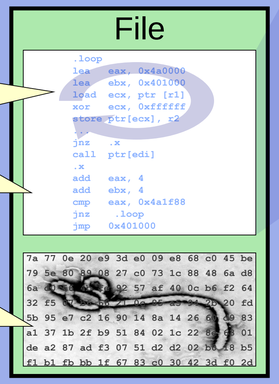
- An unpacking loop decompresses or decrypts the hidden code at runtime

Code that will be overwritten

- The code that is statically present may be modified before or after it executes

Hidden code bytes

- Hidden code is compressed or encrypted and looks like random data



During Execution

Memory image

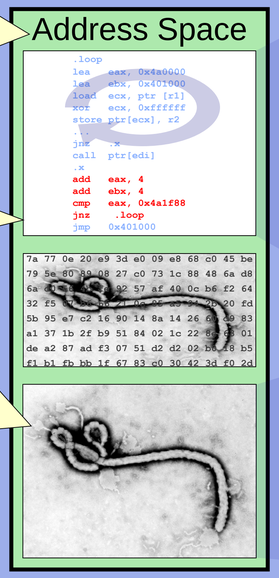
- At runtime the unpacking loop unpacks hidden code into the address space
- *But where?* We want to find the code before it executes

Modified code

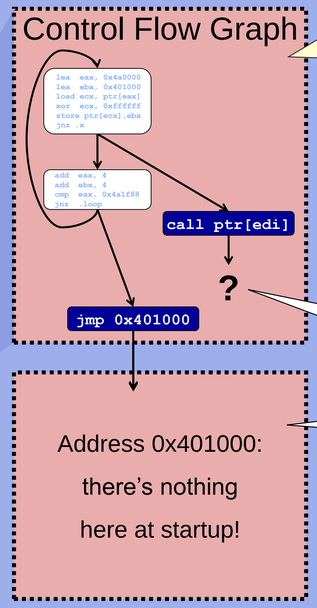
- The program's static code has been replaced by modified code

Unpacked code

- The code must be unpacked before it can be executed, though it could be unpacked a piece at a time
- At some point a control transfer instruction passes execution to an unpacked code region



Dyninst Analysis of Binary

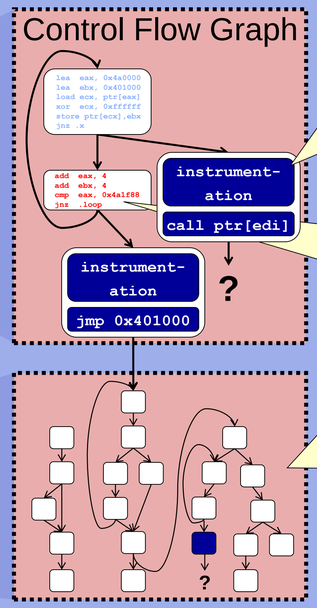


Static parsing

- Dyninst initially generates a CFG for all visible code

Obfuscated control transfer analysis

- We monitor two types of control transfer instructions:
  - Indirect transfers since we can't statically determine where they go
  - Static transfers into empty or uninitialized regions



Instrumentation

- We find new code by instrumenting control transfers to determine whether the target address is in a new code region

Modified code

- We track code writes by write-protecting code pages and capturing the resulting exceptions

Runtime parsing

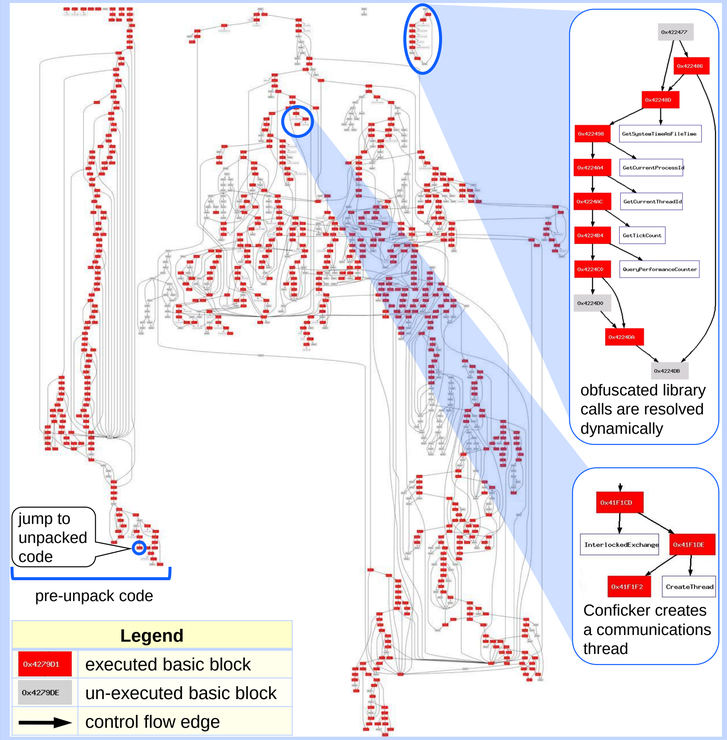
- Dyninst parses unpacked code regions just before they execute
- Candidate control transfers to other unpacked regions are instrumented

Analyzing Conficker

Conficker generates its malicious code at runtime; we use Dyninst to analyze its static *and* dynamic code.

Code coverage of basic blocks

We efficiently obtain code coverage information for Conficker A by instrumenting all of its basic blocks and removing instrumentation that has executed.



Stack traces at points of interest

We monitored communications code by instrumenting functions in the Windows socket library (e.g., **`bind, send, select`**) to walk the call stack.

Conficker's communications thread at `select`

```
Frame pc=0x7c901231   func: DbgBreakPoint  at 7x901230    [Win DLL]
Frame pc=0x10003c83   func: DYNbreakPoint  at 0x100003c70 [instrument.]
Frame pc=0x100016f7   func: DYNstopThread  at 0x100001670 [instrument.]
Frame pc=0x71ab2dc0   func: select         at 0x71ab2dc0  [Win DLL]
Frame pc=0x401f34     func: nosym1f058     at 0x41f058    [Conficker]
```