Embracing the new threat: towards automatically, self-diversifying malware

Mathias Payer <mathias.payer@nebelwelt.net>
UC Berkeley and (soon) Purdue University

Image (c) http://ucrtoday.ucr.edu/9768/assassin-bugs
Malware landscape is changing
The ongoing malware arms race

1. Generate new malware instance
2. Attack a bunch of targets
3. AV vendor gets first sample
4. Malware analysis
5. Signatures updated

The cycle continues.
Defense limitations

- Newly diversified samples are not detected
  - Basically a “new” attack

- New malware spreads fast
  - Time lag between analysis and updated signatures

- Can we automate this process?
Fully automatic diversity

*cpp

Compiler

Malware

Malware

Malware
Outline

State of the art: Malware detection

A new threat: Malware diversification

Possible mitigation: Better security practices
State of the art: Malware detection
Malware detection is limited

- Performance
  - Don't slow down a user's machine (too much)

- Precision
  - Behavioral, generic matching

- Latency
  - Time lag between spread and protection
Detection mechanisms
Signature-based detection

• Compare against database of known-bad
  – Extract pattern
  – Match sequence of bytes or regular expression

• Advantages
  – Fast
  – Low false positive rate

• Disadvantages
  – Precision limited to known-bad samples
Static analysis-based detection

- Search potentially bad patterns
  - API calls
  - System calls
- Advantages
  - Low overhead
- Disadvantages
  - False positives
  - Based on well-known heuristics
Behavioral-based detection

• Execute “file” in a virtual machine
  – Detect modifications

• Advantages
  – Most precise

• Disadvantages
  – High overhead
  – Precision limited due to emulation detection
Summary: Malware protection

• Arms race due to manual diversification
  – Signature-based techniques loose effectiveness

• Cope with limited resources
  – On the target machine, for the analysis, and to push new signatures/heuristics

• No perfect solution
  – Either false positives and/or negatives or huge performance impact
New threat: Malware diversification
Software diversification

- *.cpp
- Compiler
- Program
- Program
- Program
C/C++ liberties

- Data layout changes
  - Data structure layout on stack
  - Layout for heap objects (limited for structs)

- Code changes
  - Register allocation (shuffle or starve)
  - Instruction selection
  - Basic block splitting, merging, shuffling
Malware diversification

- Generate unique binaries
  - Minimize common substrings (code or data)
  - Performance overhead not an issue
- Diversify code and data layout
- Diversify static data as well
Implementation

- Prototype built on LLVM 3.4
  - Small changes in code generator, code layouter, register allocator, stack frame layouter, some data obfuscation passes
- Input: LLVM bitcode
- Output: diversified binary

Source: http://github.com/gannimo/MalDiv
Similarity limitations

Common subsequences in diversified binaries

Number of subsequences (log scale)

Length of subsequence

- 400.perlbench
- 401.bzip2
- 429.mcf
- 433.milc
- 444.namd
- 445.gobmk
- 450.soplex
- 453.povray
- 456.hmmer
- 458.sjeng
- 462.libquantum
- 464.h264ref
- 470.lbm
- 471.omnetpp
- 473.astar
- 482.sphinx
- perlbench vs. bzip2
- perlbench vs. gobmk
- soplex vs. omnetpp
- nmap
- simple port scanner
Demo

- Simple hello world
  - Let's see how far we can push this!

```c
#include <stdio.h, string.h>
const char foo[] = "foobar";
char bar[7];

int main(int argc, char* argv[]) {
    strcpy(bar, "barfoo");
    printf("Hello World %s %s\n", foo, bar);
    printf("Arguments: %d, executable: %s\n", argc, argv[0]);
    return 0;
}
```
Scenario 1: malware generator

- *.cpp
- Compiler
- Malware
- Malware
- Malware
Scenario 2: self-diversifying MW
Possible mitigation:
Better security practices
Mitigation

• Recover high-level semantics from code
  – Hard (and results in an arms race)
• Full behavioral analysis
  – Harder
• Prohibit initial intrusion
  – Fix broken software & educate users
  – Hardest
Conclusion
Conclusion

- Diversity evades malware detection
  - Fully automatic, built into compiler
  - No need for packers anymore
- Adopts to new similarity metrics
- New arms race between defenders and compiler writers
- Don't rely on simple, static similarity!