End-Point Counter-Worm Mechanism Using Automated Software Patching

Angelos Keromytis
Columbia University
angelos@cs.columbia.edu
Summary

• End-network architecture for protecting network services against worms (or remote attacks)

• Current model: exploit-and-repair
  ➢ Zero-day worms ?
  ➢ Forgot to patch ?

• Automatically fix software flaws

• Composition of known techniques
  ➢ Intrusion/anomaly detection
  ➢ Sandboxing
  ➢ Source-to-source transformations
  ➢ Runtime protection
Trends favoring worms

• Many software flaws
  ▶ Difficult to find, easy to "verify"

• Protection mechanisms
  ▶ Expensive, self-DoS, not deployed

• Sandboxing doesn’t help

• Fast, automated action, but no automated reaction
  ▶ Even a little reaction can help
  ▶ Software updates "sort-of" work

• Content filtering
  ▶ End-to-end, opportunistic encryption
  ▶ Polymorphism
Architectural summary

- Detect suspicious traffic
  - Worm infection vector, remote attack
  - Let the worm tell us what the flaw is
- Analyze in isolated environment
  - Detect anomaly or recognize class of attack
- Try software fixes
  - Modify source code, recompile, test
- Update production server with patched version
- All the steps occur automatically
Architecture

1. Worm Scans/Infection attempts
2. Notifications
3. Forward features
4. Vulnerability testing and analysis
5. Possible fix generation
6. Application update

Enterprise Network

Firewall Sensor

Anomaly detection analysis

Passive Sensor

Host-based Sensor

Application (e.g., Web) server

Honeypot

Sandboxed environment

Hypothesis testing and analysis

Patch generation

Instrumented Application server

Remote Sensor

Other organization
Hypothesis testing/analysis

- Sandboxed environment
  - VMWare + ProPolice
- Feed suspected exploit to instrumented application
  - Recognize known attacks (e.g., buffer overflow)
  - Anomaly detection
- Apply potential fixes, recompile, test
  - Did we break anything?
- Source availability
  - Binary rewriting
Automatic patching

• TXL for source-to-source transformations
• Various heuristics
  ▶ Move-to-heap
  ▶ Slice-off functionality
  ▶ Embedded content-filtering
Example: buffer overflow

caller() {
    char overflowed[100];
    flawed(overflowed);
}

flawed(char *buffer) {
    read(buffer);
}
Becomes...

jmp_buf worm_env;

caller () {
    char *overflowed = pmalloc(100);
    signal (SIGSEGV, worm_handler);
    if (setjmp (worm_env) == 0)
        flawed(overflowed);
}

int worm_handler () {
    longjmp (worm_env, 1);
}
pmalloc()

- Bracket allocated buffer with zero-filled write-protected pages
- Overflow or underflow causes SIGSEGV
- Signal handler can catch signal and react gracefully
Does it work?

- Looked at 17 vulnerable applications assembled by CoSAK project
- Could fix 14 of them out of the box
  - Less than 10 seconds for Apache
- Can probably fix 1 more
- Performance impact minimal (with some extensions)
- Many details left out
  - sizeof(), static declarations, multi-dimensional arrays, ...
Observations

• Let the worm reveal the vulnerability
• Apply expensive fixes in a localized manner
  ▶ "Knowledge" of the code helps
• Deploy per network, managed security company, ...
  ▶ Share infection-vector (CCDC ?)
• No need to trust someone else’s patches
  ▶ No WAN dependency
Limitations and future work

- Source code availability
- Generalization
  - Application-level DoS
- Correctness of patched binary
  - Human in the loop?
- Server still gets compromised
  - Combine with lightweight protection mechanisms
- Better analysis tools and heuristics
- Better ways to target patches
  - Aspect-oriented programming, ...
- Email worms

• Other related work on worms

• angelos@cs.columbia.edu