Enhancing Botnet Resitence with Deterministic Execution

Bryan Ford (PI)
Amittai Aviram
Yale University

ONR Botnets Meeting – February 9, 2010
Why Botnets Exist

In today's information society, the purpose of most botnets is to steal information:

- Credit card numbers
- Personal information useful in identity theft
- Corporate or government secrets
- Information useful in committing other crimes: e.g., building plans, access codes, …
Approaches to Botnet Defense

- **Network-based detection and response:**
  - Mitigate statistically discernible attacks
  - But too late – *some* information already gone
  - “Stealth botnets” can mimic/use normal traffic

- **Network-based prevention:**
  - Block attack traffic from active adversaries
  - But botnets can trick clients into “consenting” to visit a malicious/hacked web page

⇒ Host-based prevention also important
Current Host-based Defenses

- Most work focuses on security bugs and protection mechanisms
  - Detect or prevent buffer overruns, etc.
  - Make attacks likely to fail (e.g., randomization)
  - Prevent buggy code from leaking secrets (IFC)
- But botnets can steal critical information without compromising protection mechanisms!
Unintended Information Channels

Botnets can (and will) exploit any information they can get about a target, however “harmless”

Two example scenarios:
1. Exploit host system information to evade IDS
2. Exploit timing channels to steal secrets
Scenario 1: The Setup

Acme Inc deploys *honeyfarm* to detect attacks

- “Real” client hosts interact with the Internet realistically, e.g., running web access traces
- ...but run under a VMM for safety, analysis

Honeyfarm PCs purchased all in one batch

- Homogeneous hardware platform: HP, Intel

But corporate network is heterogeneous!

- *oops*
Scenario 1: The Attack

- Attacker learns/guesses that honeyfarm is homogeneous, e.g., all Intel CPUs
- Compromised web servers use JavaScript to get host information from potential victims
- JavaScript reports Intel vs AMD, server returns attack code only if AMD
  - What if JavaScript doesn't give CPUID info?
  - No problem, just call: sin(-19153168)

Lesson: *any* host system info may be sensitive
Scenario 2: The Setup

- Acme Inc employees need to browse the web to do business, watch competitors, etc.
- All client hosts have “red/black” protection
  - **Red VM**: runs E-mail, browsers, etc.
  - **Black VM**: holds critical company secrets; runs only trusted, centrally approved apps
Scenario 2: The Attack

- Attacker's compromised web server supplies JavaScript/Java/Flash/NativeClient code
- Client-side attack code runs in Red VM, but uses timing capabilities to attack Black VM
  - extract private SSL keys [Percival, Brumley…]
- JavaScript too slow? Try JIT, NativeClient

**Lesson:** even *time* can be sensitive information
Solution: Deterministic Virtualization

Harden end hosts by running untrusted code in a deterministic virtual machine

- *All* behavior is precisely software-defined
- *No* variation based on host platform
  - CPUID instruction yields fake/generic info
  - \( \sin(-19153168) \) yields same thing on Intel/AMD
- *No* variation based on execution timing
  - gettimeofday() yields synthetic/sanitized time
  - eliminate all implicit timing dependencies

No nondeterministic inputs → no side channels
Challenges

Determinism is not easy to achieve:

- CPUs leak host information in myriad ways
  - CPUID or browser/OS information
  - floating-point rounding errors on sin, exp, etc.
  - x86: undefined condition codes after MUL, DIV
- Multithreaded code is highly timing-dependent
  - can disable/virtualize gettimeofday(), but
  - any thread can always spin to time another
Deterministic Sandboxing Goals

Fully isolate untrusted guests (e.g., Java, JavaScript, native) from host platform

- **Protection**: use conventional techniques to confine all guest actions to sandbox
- **Platform isolation**: hide all platform-specific host behavior via selective instruction rewriting
- **Timing isolation**: hide all timing information via deterministic parallel programming model
Platform isolation

To hide platform-specific behavior:

- *All code*: see only generic/artificial platform
- *All code*: use only generic software-based transcendental math (e.g., sin, cos, exp, log)
- *Native code*: rewrite all instructions with undefined/unpredictable results or behavior
  - x86: shift/rotate, mul/div, bit test/set/reset... all leave “undefined” condition code flags
Vx32: Lightweight Code Rewriting

Runs native x86 code
- Unmodified binaries (Linux, Plan 9)
- Data sandboxing: x86 segments
- Code sandboxing: rewrite to safe cache
- Rewrite sensitive instructions
Vx32 Efficiency on Compute Tasks
Timing Isolation

Not enough to disable or virtualize timers: *any* nondeterminism leaks time information

- Single-threaded code can be deterministic, but parallel code is generally nondeterministic
- Java is already multithreaded – *everything* will be soon

Multicore Revolution \[\times\] Web Applications

- In standard nondeterministic thread models, *any* thread in a spin loop is a timer
Deterministic Parallelism

New memory model: deterministic consistency

• Deterministic thread synchronization
  – fork/join, barriers, futures, pipelines, …
  – not mutual exclusion!

• Deterministic memory access model
  – all reads/writes normally thread-private
  – reconcile changes at synchronization points

➔ Flexible, scalable, and timing-oblivious
Deterministic Consistency (DC)

access orderings possible under sequential consistency

deterministic consistency

thread_fork

x := y
y := x

W y
R x
W x
W y
R x
W x

swap x and y

Behaves like parallel assignment statement: x, y := y, x
[Perl, Python, Ruby, JavaScript, ...]
Implementing DC in vx32

- Threads run in *separate* vx32 sandboxes
- Use VM, DSM techniques to copy & reconcile

```
\begin{align*}
x & := y \\
1 & 2 \\
x & := x \\
2 & 2 \\
y & := x \\
1 & 1 \\
\end{align*}
```

- Alternative approaches: custom hardware, ...
Results So Far

- Vx32 prototype with deterministic consistency
  - Runs a few parallel benchmark apps – some fast enough, big performance hit on others
  - Still many limitations, protection incomplete...

- In progress:
  - OpenMP-compatible parallel environment
  - Efficient kernel-level implementation
Conclusion

- Network-based and host-based botnet defense *must* go hand in hand to be effective
- Standard focus on *protection mechanisms* overlooks sensitivity of host, timing information
- Proposal: *deterministic virtualization* can close unintended information channels, thwart “under-the-radar” stealth botnets
  - *Platform-oblivious*: can't exploit platform info
  - *Time-oblivious*: can't exploit timing channels