Correlating Low-Level Events To Identify High-Level Bot Behaviors

Liz Stinson
John Mitchell
Stanford University

Matt Fredrikson
Somesh Jha
University of Wisconsin

Lorenzo Martignoni
University of Milan
Our anti-inspirations

• “Personal firewalls”: identify when an app is connecting to the network

  Too ambiguous

• Host-level methods that inundate us with information (all registry accesses/changes, file accesses/changes) without providing a higher-level assessment of what’s going on

  Too noisy; devoid of meaning
Problem Statement

• >5M “distinct, active” bot-infected machines detected between January - June, 2007
  – “active”: carried out at least one attack
  – Symantec Threat Report, Volume XII
• The *best* anti-virus signature scanners fail to detect anywhere from 30% to 50% of malware samples seen in the wild
  – NB: The best AV scanners may not be who you think they are…
Problematic Asymmetry

Malware writers know they have the advantage here and they exploit it.

Tens of thousands of novel malware variants created annually

AV companies decide which undetected malware to create sigs for using triage; must exceed some prevalence threshold.

Malware writers know they have the advantage here and they exploit it.
Existing behavior-based detection

- Identify simple, mostly stateless "features" (process execution characteristics); e.g.
  - Which dir(s) does app live in? write to? App = shadow?

- Traits malware have adapted to evade AV detect
- Statefully scan network packet contents
- More general characterizations
  - Abstract: spyware monitors/reports user actions
  - Concrete: rootkits that load kernel modules

- May identify *incidental*, rather than *fundamental* behaviors

- For ML-based approaches, may be other ways to achieve same end (i.e. ways not included in model)
Broad spectrum. How to evaluate?

- How *effectively* does this method distinguish malicious behavior from benign?
- How *thoroughly* is target behavior captured?
- How *complex* is the identified behavior?
- How *fundamental* is the behavior to the malware’s purpose?
Goals

- We want to identify high-level behaviors
  - “downloading and executing a program”
  - “acting like a TCP server”
  - “acting like a proxy”
  - “leaking sensitive data”

- Bot-command-level actions
  - Via monitoring process execution
  - Distinguish malicious from benign instances of above by identifying if remotely initiated

Sample bot commands

- `http.execute <URL> <local_path>`
- `harvest.registry <reg_key>`
- `redirect <lport> <rhost> <rport>`
- `startkeylogger`
Example: Acting like a proxy
Identifies ordering dependencies

tcp_client

net_recv

&

tcp_proxy

tcp_server

net_send

Not shown here
edge constraints
die operations
socket duplication
intervening irrelevant ops
Including parameters and constraints

\[
\text{tcp_client}(sd0, \text{rem_ip, rem_port}) \quad \text{tcp_server}(sd1, \text{loc_ip, loc_port, cli_ip, cli_port})
\]

\[
(sd2 == sd0) \quad (sd3 == sd1)
\]

\[
\text{net_recv}(sd2, \text{recv_buf}) \quad \rightarrow \text{recv_len}
\quad \text{net_send}(sd3, \text{send_buf}) \quad \rightarrow \text{send_len}
\]

\[
(\text{recv_len} > 0) \quad (\text{send_len} > 0)
\]

\[
(\text{send_buf} == \text{recv_buf})
\]

\[
\text{tcp_proxy}(sd, \text{loc_port, rem_ip, rem_port})
\]

Constraints can be pre-conditions or post-conditions
We'll focus on this

\[
\begin{align*}
\text{net\_send(sd3, send\_buf)} & \rightarrow \text{send\_len} \\
\text{net\_recv(sd2, recv\_buf)} & \rightarrow \text{recv\_len} \\
\text{send\_len > 0} & \land \text{recv\_len > 0} \\
\text{send\_buf == recv\_buf} & \\
\text{tcp\_proxy(sd, loc\_port, rem\_ip, rem\_port)}
\end{align*}
\]
(send_buf == recv_buf)

- Too constrained; really want to express: the buffer that is sent is derived from a buffer that is received
- Augment (add action to): on_match of net_recv
  
  set_tainted( recv_buf, sd2 /*taint label*/ )

- Change condition to:
  
  tainted( send_buf, sd2 /*taint label*/ )
Modified graph

... → net_recv(sd2, recv_buf) → recv_len

... → on_match

net_send(sd3, send_buf) → send_len

(send_len > 0) → set_tainted(recv_buf, sd2)

&

tainted(send_buf, sd2)

tcp_proxy(sd, loc_port, rem_ip, rem_port)
.redirect <loc_port> <rem_host> <rem_port>

Add constraints

tcp_client(sd0, rem_ip, rem_port)

net_recv(sd2, recv_buf) -> recv_len

tcp_server(sd1, loc_ip, loc_port, cli_ip, cli_port)

net_send(sd3, send_buf) -> send_len

tainted(rem_ip) && tainted(rem_port)
tainted(loc_port)

&
tcp_proxy(sd, loc_port, rem_ip, rem_port)
“Language” our system exports

- Set of high-level primitives that can be combined to describe interesting behaviors
  - tcp_client, tcp_server, net_send, net_recv, create_exec_file, ...
- Using these, we can detect:
  - Leak private data (reg key values, file contents, system info, …)
  - Download and execute a program
  - Send email
  - Proxy
  - Keystroke logging
Challenges

• Posed by proprietary-OS environment
  – Opacity; identifying operations & constraints
  – Replicating OS semantics
• Posed by syscall interposition generally
• Posed by hypothetical attempts to evade
  – Split behavior across processes or across runs of the same application
  – Expropriate kernel functionality
    • e.g. raw sockets
Summary

- Target the behaviors that make bots useful
- Identify the essential ops in those behaviors
- Use data-flow analysis info variously
- Good initial results against bots
  - Including: rbot, agobot, dsnxbot, spybot, ...
  - Use bot commands as inspiration
  - Resilient to encryption of bot communications
- Good initial results against benign progs
  - When testing against specifications that encode remote-control requirement
  - Performing user-input tracking