Fides: Remote Anomaly-Based Cheat Detection Using Client Emulation

Ed Kaiser
Wu-chang Feng
Travis Schluessler
The Cheating Problem

• Networked games simulate complex worlds
  • limited server computation
  • player sensitivity to network latency

• Client is trusted to run simulation locally
  • follow game rules
  • keep secrets from player

• Cheats are software that abuse the trust
  • accomplish feats a cheater is unable to do
  • automate actions a cheater is unwilling to do
Cheating Affects Online Games

- Frustrates legitimate players
  - distorts in-game economy
  - not fun to play against cheaters
  - cannot tell if good opponents are legitimate
    - good players get accused of cheating

- Impacts profitability of game developer
  - existing players may quit in frustration
  - bad reputation dissuades new players
  - increases operating costs
    - EVE online: 2% bots → 30% of load
What Cheats Modify [NG08]

- **Game memory** via \texttt{WriteProcessMemory()}:
  - static data (e.g., gravity constants)
  - dynamic data (e.g., location, health, team)
  - altering existing code (e.g., *hot patch*)
  - injecting new code (e.g., *DLL injection*)

- **Game execution**
  - thread hijacking (e.g., *detour*, *function hooking*)
  - thread injection via \texttt{CreateRemoteThread()}
  - as debugger via \texttt{DebugActiveProcess()}
**Nemesis Warcraft III MapHack**

- Reveals map and secret opponent locations
  - inject DLL containing cheat code
  - detours rendering functions
  - hooks I/O handlers for toggling operation
  - calls game message functions to display status

---

**Cheat Enabled**
State-of-the-Art in Defense

- Signature-based cheat detection
  - generate cheat-specific signatures
    - must obtain working cheats
    - continual developer effort
    - state grows as new cheats are cataloged
    - does not deal well with polymorphism
  - search **every** process for known signatures
    - indiscriminately reads private data (e.g., Blizzard’s Warden)
    - prone to false positives:

  - “Rifle Aim Prediction:
    - into IRC channel
  - Tricking Punkbuster
  - Tricking VAC
Similar to other Security Problems

• Similarity to rootkits
  • adversary controls the machine
    • has administrator privileges
    • runs before anti-cheat software
    • can modify the operating system and other tools
  • uses advanced techniques
    • cloak itself just-in-time (e.g., Hoglund’s Supervisor)
    • spoof anti-cheat software results

• Similarity to viruses
  • obfuscation to prevent reversing
  • polymorphism to thwart signature detectors
... yet is Distinct Security Problem

- Adversary is owner of machine
- Always connected, for long time periods
  - cannot disable server-initiated security (unlike Windows Update)
  - server can perform arbitrary checks on-demand
- Typically targets game code
  - limited places to attack
  - can do anomaly-based detection (à la kernel integrity approach)
- Presence is not immediately catastrophic
  - can wait to take action (to prevent cheater from learning)
  - machine is not used to attack network hosts
  - damage can be rolled back easily
    - unlike reissuing stolen credit card numbers, SSNs, etc.
- Monetary penalty for being caught
  - $50 game copy, $10/month, plus time lost (opposed to botnet)
The Fides Approach

• Approach leverages properties of problem

• **Anomaly-based** cheat detection
  • know what game looks like
    • finite state
    • readily available
  • search the game client for deviations
    • cheater targets game code
    • cheat agnostic
  • detection is sufficient
    • cheat is not immediately catastrophic
Fides Approach cont’d

• Via *continued random remote measurement*
  
  • unpredictable
    • conceals what will next be measured and when, instilling “fear-of-the-unknown”
  
  • probabilistic
    • no need for complete integrity check
    • server has indefinite contact with the client
  
  • continued
    • can audit until absolutely confident in detection
Fides Approach cont’d

• Using *partial client emulation*
  • to accommodate client system variation
    • between players
    • between sessions (e.g., desktop vs. laptop)
  • regarding libraries, versions, and locations
  • always connected, for long time periods
Fides Architecture

- Controller decides **what** & **when** (and eventually **how**) to measure the client
  - compares measurement to emulated state
  - alters player account when caught cheating

- Auditor **only reads** game process state
Limitations to Approach

• As a software implementation
  • Auditor will become target of attack
    • feed it known legitimate client process-state data
      • statically generated (pre-sampled and stored in database)
      • dynamically generated (running second unmodified client)
    • hook it to know when to unload cheat

• Cannot catch cheats external to game client
  • collusion cheats
    (e.g., online poker cheaters)
  • robotic cheats
    (e.g., Guitar Hero robot)
Addressing the Limitations

• Auditor is the weak point
  • cryptographically entangle the Auditor
    • similar approach to Pioneer
  • use secure hardware to verify Auditor operation
    • similar to Intel anti-virus presence detector
  • run the Auditor itself on secure hardware
    • (e.g., the Intel AMT Manageability Engine)
    • similar to Copilot

• Does not detect completely external cheats
  • anomaly-based detection on user behavior or
    statistics, available to server
Controller Design

- Emulates client-process state
  - drives audit strategist (could be game-specific)
  - used to validate audits
Partial Client Emulation

• Done once at client login
  • share library names, versions and locations

• Works on *any* commercial-off-the-shelf game

• Binary Parser recreates client layout
  • uses libraries/executable stored at Controller
  • uses client layout (base locations) to map them
  • identifies and hashes static code & data
    • high confidence in client understanding
  • identifies dynamic data regions
    • more expensive (i.e., game-specific) to validate
    • high confidence in client understanding
Partial Client Emulation cont’d

- Code Disassembler
  - creates a rough call graph
    - learns instruction range for each function
    - learns **CALL** addresses (i.e., relating functions)
    - lower confidence (difficult to get complete coverage)

Partial Call Graph of a Homebrew game
Partial Client Emulation cont’d

• Execution Sampler and Execution Profiler
  • run game in VM with identical layout to client
  • learn dynamic calls not obtainable through static analysis
    • use hardware debugging techniques to single-step through execution
Auditor Design

- Measures the client process
- returns the data to the controller
Evaluation

- Implemented Auditor & Controller in C++
- running on separate 2.39GHz Intel Core2
- on commercial-off-the-shelf game Warcraft III

<table>
<thead>
<tr>
<th>Task</th>
<th>Cycles</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample Memory</td>
<td>38,000</td>
<td>15.9µs</td>
</tr>
<tr>
<td>Hash Page</td>
<td>113,000</td>
<td>47.2µs</td>
</tr>
<tr>
<td>Trace Stack</td>
<td>64,400</td>
<td>26.9µs</td>
</tr>
<tr>
<td>Detect Debugger</td>
<td>23,200,000</td>
<td>9.7ms</td>
</tr>
<tr>
<td>Parse All Binaries</td>
<td>236,000,000</td>
<td>98.8ms</td>
</tr>
<tr>
<td>Disassemble Code</td>
<td>205,000,000</td>
<td>85.9ms</td>
</tr>
<tr>
<td>Validate Hash</td>
<td>3,170</td>
<td>1.3µs</td>
</tr>
<tr>
<td>Validate Stack</td>
<td>10,800,000</td>
<td>4.5ms</td>
</tr>
<tr>
<td>Validate Debugger</td>
<td>130</td>
<td>52.4ns</td>
</tr>
</tbody>
</table>
Evaluation Timeline

• Learn newly patched game
  • account for security (i.e., obfuscation & anti-debugging)
• Acquire and run first-to-release cheats
  • verify correct operation
• Detection!

WC3 patch 1.22b
October 29, 2008

Nemesis cheat released & detected by Fides
December 12, 2008

Fides works on WC3 & detects simple new cheats
November 10, 2008
Experiment Setup

- Ran the following cheats:
  - Bendik’s MH, NOPs a few bytes
  - Kolkoo’s MH, NOPs bytes over more pages
  - Revealer MH, NOPs and hooks input functions
  - Simple MH, NOPs bytes over many pages
  - Nemesis MH, complex and “undetectable”

- Periodically audit (±5% randomness)
- Code page hash audit the game
  - hash currently executed code page
- Measure the mean audits required to detect
  - averaged over 1000 trials
Results

• Auditing roughly once every 100ms

<table>
<thead>
<tr>
<th>Cheat</th>
<th>Avg Audits Required</th>
<th>Wall-clock Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bendik’s MH</td>
<td>1265.3</td>
<td>2min 08.4s</td>
</tr>
<tr>
<td>Kolkoo’s MH</td>
<td>733.4</td>
<td>1min 14.5s</td>
</tr>
<tr>
<td>Revealer MH</td>
<td>309.8</td>
<td>31.4s</td>
</tr>
<tr>
<td>Simple MH</td>
<td>260.4</td>
<td>26.4s</td>
</tr>
<tr>
<td>Nemesis MH</td>
<td>204.1</td>
<td>20.5s</td>
</tr>
</tbody>
</table>

• Auditing roughly once every second

<table>
<thead>
<tr>
<th>Cheat</th>
<th>Avg Audits To Detect</th>
<th>Wall-clock Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bendik’s MH</td>
<td>1309.7</td>
<td>21min 49.7s</td>
</tr>
<tr>
<td>Kolkoo’s MH</td>
<td>733.0</td>
<td>12min 13.0s</td>
</tr>
<tr>
<td>Revealer MH</td>
<td>322.2</td>
<td>5min 22.2s</td>
</tr>
<tr>
<td>Simple MH</td>
<td>301.3</td>
<td>5min 01.3s</td>
</tr>
<tr>
<td>Nemesis MH</td>
<td>264.1</td>
<td>4min 24.1s</td>
</tr>
</tbody>
</table>
Detecting Warcraft III MapHacks

![Graph showing the average audits to detect various MapHacks over different auditing periods. The graph includes lines representing Bendik's MapHack, Kolkoo's MapHack, Revealer MapHack, Simple MapHack, and Nemesis MapHack. The x-axis represents the auditing period (log10 ms), and the y-axis represents the average audits to detect.]
Conclusions

• Cheats are advanced
  • large range of cheat methods
  • present a distinct security problem

• Fides is specifically designed to catch them
  • anomaly-based detection
  • via continued random remote measurements
  • using partial client emulation
Thanks