Software Integrity Protection Using Timed Executable Agents

Juan Garay, Lorenz Huelsbergen
Bell Labs, Alcatel-Lucent
{garay,lorenz}@research.bell-labs.com
Motivation

- Malware is ubiquitous and on the rise
  - Spam
  - Viruses
  - Trojans
  - Improper reconfiguration
  - Keyboard loggers
  - ...

- Pressing need to safeguard a system’s integrity
- Many deployed systems contain little or no hardware protections
- Our goal: Develop off-the-shelf software protections
  - Perhaps not 100% secure, but
  - raise the bar substantially
Software-only Integrity Protection

- Software protection mechanisms
  - Can often detect if malware is present on a system
  - Can make it difficult for present malware to function
  - Requires no special hardware support
  - Ideal for legacy systems that have become vulnerable
  - Supports hierarchical integrity checking
Approach: Software Agents

- Mobile code
- Typically traverses a network to carry out tasks (secure deployment requires authentication)
- If run on a known platform, work (execution time) done by an agent can be monitored
- Useful in many scenarios
  - Including system integrity checking
TEAS: Basic Idea

Challenge: Agent carries a program to run (many agents)
Response: Program result + side effects

```
for(a=start; a<end ; a++)
    sum+= *a;
return sum;
```

Bounds on execution time can help deter adversary’s analysis of the challenge program(s)

Important: Agent may be an arbitrary program
Themes

• “Cryptographic time capsules:” Sending information into the future

Important: Verifiability (of contents, “time” parameters)

• Moderately-hard functions: Not computationally infeasible to solve, but also not easy
Talk Outline

- Definitions, assumptions and system requirements
- TEAS solutions for
  - Offline Adversaries
  - Online Adversaries
- TEAS applications
- Related work
- Summary
Def’s, Assumptions & System Requirements

Model:
- Collection of computational nodes in a network
- Two types of nodes:
  - Secure/Trusted hosts
  - Insecure clients
- Uniprocessors. CPU rate $C, [C] = \text{cycles/sec}$
- Memory: Code, data, unused memory and program stack
- Communication: Fairly accurate estimate of transmission delays
  Bandwidth $B, [B] = \text{bits/sec}$
Adversaries

- **Goal:** Provide defense against client nodes being corrupted and/or taken over by an attacker – the *adversary*

- Two adversary classes:
  - *Offline adversaries:* Adversary tries to analyze incoming programs (“agents”) *without running them* – recall *static analysis*
  - *Online adversaries:* Adversary is able to run incoming programs

- **Assumption:** Adversary makes *no changes* to the client’s hardware

- Client’s computing power is known
  \[\Rightarrow \text{no. computation steps} \approx \text{absolute time}\]
TEAS Definitions

\((\varepsilon, \mathcal{A})-TEAS = (T_{gen}, T_{ver})\)

(Two probabilistic algorithms, run at \emph{Challenger})

\[ T_{gen}(\text{params}) \rightarrow T = ( (P_1, o_1, t_1, \pi_1), \ldots, (P_k, o_k, t_k, \pi_k) ) \]

\[ t_i = |P_i|/B + |o_i|/B + D(P_i)/C \]

\[ \pi_i : \text{"patience" threshold} \]

\[ P_i \rightarrow (o_i', t_i') \) (run at \emph{Responder}) \]

\[ T_{ver}( T, (o_1', t_1'), (o_2', t_2'), \ldots, (o_k', t_k') ) \rightarrow \{\text{OK, ¬OK}\} \]

If \emph{Responder} is corrupted (by adv. in \( \mathcal{A} \)), then probability that Tver outputs OK < \( \varepsilon \)
System Requirements

- **Known *a priori***:
  - The (valid) software that runs on the Responder
  - Responder’s HW configuration (memory size, processor speed)

- Ideally, Challenger and Responder are connected by a deterministic, tightly coupled, network with known latencies. (Also more loosely coupled networks.)

- OS: Responder OS allows full and uninterrupted access (i.e., disable interrupts, time-slicing, etc.). Provisioned to receive and execute agents. Examples: real-time computing OSes (mobile phones, computing “appliances,” etc.)
Talk Outline

- Definitions, assumptions and system requirements
- TEAS solutions for
  - Offline Adversaries
  - Online Adversaries
- TEAS applications
- Related work
- Summary
TEAS for Offline Adversaries

- *Offline adversary* tries to analyze incoming programs *without running them*. Also access to inputs and state of Responder.

- **Approaches**
  1. Undecidability-based protection
  2. (Program analysis) Complexity-based protection
Complexity-based Protection

- **Program analysis background:** Behavior of program P (output values) may be determined through *global data-flow analysis*
  - Extract P’s *control flow graph* (CFG) $G_P$
  - Convert $G_P$ to a *reducible* flow graph $G'_P$
  - Perform global data flow analysis on $G_P$ (or $G'_P$)

- Let $n$ be some static measure of $\text{IPI}$
  - Extraction of CFG has complexity $\Omega(n)$
  - Rises to superlinear ($\Omega(n^2)$ or higher) with certain types of branches
  - CFG may not be reducible
  - Note: Only *deterministic* program analysis [Gulwani et al.]
Complexity-based Protection (cont’d)

- Program $P: P(x) \rightarrow y$, fast execution time
  - E.g., check memory locations, configuration values
- Global Data Flow problems ($\Omega(|P|^2)$ worst case)
  - “Reaching Definitions (Def-Use):” For each use of a variable, determine all the definitions that reach that variable
- $\Rightarrow$ Analysis (much) more expensive than execution
- Since we are crafting the agent, it is possible to avoid “in-practice” analysis!
- **Strategy:** Generate TEAS instance with sufficiently many such programs ($k$)
Complexity-based Protection: Example

\( C = 10^9 \) cycles/sec, 1 instruction/cycle; \( B = 10^6 \) bytes/sec

\(|P| = 10^3\) instructions, 4 bytes/instr.

Linear dynamic runtime \( \Rightarrow \frac{10^3}{10^9} = 0.000001 \) sec

Communication \( 4 \times \frac{10^3}{10^6} + \frac{4}{10^6} = 0.004004 \) sec

\( \Rightarrow t \geq 0.004005 \) sec

\( \Omega(n^2) \) analysis \( \Rightarrow 0.01 \) sec computation time

\( \Rightarrow t' \geq 0.014005 \) sec

\( \approx 3.5 \)
TEAS Agent Creation

- Need a large library of agents
  - To prevent agents being “learned” by adversary
- Creation of agents by hand is possible, but tedious and error prone
- Can agents be created automatically?
  - YES, via *program blinding*
Automatic Agent Generation

- **Program blinding**: Combine a small (hand-written) program with a random, obliviously generated one
  
  \[ P^* \leftarrow P \otimes P_R \]

  “cross-over” operation: \( P^* \) “inherits” some of \( P \)’s properties

- \( P^* \) is difficult to predict:
  
  \((\varepsilon, n)\)-semantically uncertain: given \( P \) and input \( x \), \( \mathcal{A} \) can’t determine \( y \leftarrow P(x) \) after \( n \) steps of analysis with prob. better than \( \varepsilon \)

- *Input-sensitive* blinded programs
Automatic Agent Generation (cont’d)

- Make sure that blinded programs are
  1. “hard,” e.g., contain an irreducible CFG, and
  2. “input-sensitive,” i.e., blinded program’s output depends on original program’s input (e.g., a register or memory location value)

- **Experiments:** VRM, 8 registers, \( P = \text{LOAD}_r0([A]), \ r1 \leftarrow P \)
  Ran blinding \( 10^5 \) times for programs of size 25, 50, 100

<table>
<thead>
<tr>
<th># instructions</th>
<th>( n )</th>
<th>( n^2 )</th>
<th>( n^3 )</th>
<th>forward jmps</th>
<th>backward jmps</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>687</td>
<td>48</td>
<td>1</td>
<td>2.5</td>
<td>1.9</td>
</tr>
<tr>
<td>50</td>
<td>422</td>
<td>59</td>
<td>1</td>
<td>5.3</td>
<td>4.2</td>
</tr>
<tr>
<td>100</td>
<td>282</td>
<td>26</td>
<td>1</td>
<td>10.7</td>
<td>9.6</td>
</tr>
</tbody>
</table>
Example TEAS Construction

- $p_{\text{irred}}^n$: probability that random program of size $n$ contains an irreducible CFG (estimated in several ways)
- $T_{\text{gen}}$ generates instances that include $((1 - p_{\text{irred}}^n), n)$-uncertain agents
- $(\varepsilon, A_{\text{off}})$–TEAS: $T_{\text{gen}}$ blinds target program with $k$ terminating random programs s.t.
  1. every program is input-sensitive, and
  2. $k$ such that $(1 - p_{\text{irred}}^n)^k < \varepsilon$

$T_{\text{ver}}$:

if $\exists P_i^*$ s.t. $o_i \neq o_i'$ OR $t_i' / t_i > \pi_i$ then output $\neg \text{OK}$
Talk Outline

- Definitions, assumptions and system requirements
- TEAS solutions for
  - Offline Adversaries
  - Online Adversaries
- TEAS applications
- Related work
- Summary
The "interpreter" attack: Dynamic (runtime) interpretation of an agent

```c
/* agent fragment:
    LOAD (0xfc00), r0;
    STORE r0, (0x1200);
    JMP 0x0100;
*/

instr = GET_INSTR(pc);
op1 = GET_OP1(pc);
op2 = GET_OP2(pc);

switch (instr) {
    LOAD:   if (protected(op1))
            op2 = *(translate(op1));
            else
                op2 = *op1;
                break;
    STORE:  if (protected(op1))
            *(translate(op1)) = op2;
            else
                *op1 = op2;
                break;
    JMP:    if (protected(op1))
            pc = translate(op1);
            else
                pc = op1;
                break;
    ...
}
```

Loads and stores from/to protected areas are avoided, with very small overhead.
TEAS for Online Adversaries (cont’d)

Strategy: Perform “adversary fragmentation”
Force the adversary to relinquish control of the client, or not be able to respond to queries in a timely manner
"Adversary Fragmentation"

\[(1/|M|, \mathcal{A}_{on})-\text{TEAS} = (P_1, P_2, P_3)\]

\[P_1 = \text{random permutation of memory}\]

```
a2 = seed;
for(i = 0; i < N; i++) {
    a1 = random(a2);
    a2 = random(a1);
    t = M[a1];
    M[a1] = M[a2];
    M[a2] = t;
}
```
Adversary Fragmentation (cont’d)

\( P_2 = \) random queries (\( O(1) \) time)

If \( \mathcal{A} \) executes \( P_2 \), then
Responder under control of \( P_2 \)

Otherwise, \( \mathcal{A} \) computes perm. for each query
  \( \Rightarrow \) time bound violation

or tries to keep track of addresses
  \( \Rightarrow \) out of memory!

\( P_3 = \) restores original state

\( (r_1, r_2, \ldots) \)
TEAS Applications

Monitor integrity of

- Mobile devices such as cell phones
  - Has phone been hacked?
- Set-top boxes and cable modems
  - E.g., detect reconfiguration to bypass service license
- Wireless basestation components
  - Detect black/grey market cards or reconfigurations
- Remote sensors perhaps deployed in hostile environments
  - Ascertain veracity of data
Related Work

Other software-only schemes for integrity verification:

- **Genuinity** [Kemell-Jamieson, Usenix’03]: Checksum of (virtual) memory addresses and machine-specific register values; host also computes the checksum and times the response.

- [Shankar-Chew-Tygar, Usenix Security’04]: Genuinity is vulnerable to fast simulation (“interpreter”) attack, below 35%.

- **SWATT** [Seshadri-Perrig-van Doorn-Khosla, ISCC’04]: Also checksum of probabilistic memory traversal. Focuses on embedded microcontrollers, with fixed processor speeds and small memory sizes; requires knowledge of entire state being checked, and tight coupling between host and client.
Summary

- Malware is pernicious
  - Degrades and destroys system integrity
- A new method to help stop it: TEAS
  - Software-only
    - Amenable to legacy systems that might now be vulnerable
  - Challenge/response framework
    - Challenges are arbitrary programs sent as agents
    - Hard problems from complexity of program analysis
    - Challenges are timed
- New technique called program blinding
  - Aids in creating large libraries of agents
- Many application areas identified
References


Available from

http://www.bell-labs.com/user/garay
Software Integrity Protection Using Timed Executable Agents

Juan Garay, Lorenz Huelsbergen
Bell Labs, Alcatel-Lucent
{garay,lorenz}@research.bell-labs.com
Questions and Answers
Undecidability-based Protection

- Use undecidability of non-trivial program properties (Rice’s theorem)
- **Example:** Compute the number of instructions a TEAS agent executes – non-computable *a priori*
- **Challenge:** Automatic methods for generating agents with given undecidability properties