Intrusion Detection

- Define attacks using a **signature**
  - This is just a pattern on events/actions

- Three categories
  - Network Based
    - Inspect raw network packages
  - Host Based
    - Software that takes advantage of OS facilities
  - Stack Based
    - Integrated with the TCP/IP stack (vendor specific)
Network Intrusion Detection Systems (NIDS)

• Purpose
  – Detect an intrusion coming from the network

• Current Solutions (sketch)
  – Define attack as an attack signature
  – Match attack signature with ongoing activities

• How
  – Regular expression over events
  – Attack signatures capture a whole class of attack instances
Snort

- Snort
  - Preprocessor (after package decode)
  - Rule matching
  - Output (alerts, logs, counter measures)

- For example

```plaintext
alert tcp any any -> 192.168.1.0/24 111
  (content:''|0 01 86 a5|'';
  msg:''mountd access'';)
```
Problems

• Coming up with an attack signature
  – Analysts inspect examples
  – Hypothesize about the properties that must hold
  – Write down the expression

• No systematic way to
  – check for false positives or false negatives
  – evaluate the impact of attack signature changes
GARD

- Session Signatures
  - The entire attack as a regular language
- Attack invariant
  - Another representation of the attack, used to evaluate session signatures
- Semantic model of attack protocol
  - Finite state machine
    - How protocol commands alter protocol state
- Generation, Analysis, Refinement, Deployment
Systematic Method

(1) Initial session signature (syntactic features)
(2) Attack invariant (semantic features)
(3) Compare (1) with (2)
   • If false positives or false negatives go to (1), else exit
Using an example

- Ftp-cwd attack (BlackMoon FTP server)
  - Login (anonymous)
  - Send cwd command with an overly long argument, will cause a buffer overflow.
Signature Specification

• Based on 3 parts
  – Preparation
    • Attacker sets up the attack's pre-conditions
  – Exploitation
    • Attacker launches the attack
  – Confirmation
    • Attacker determines that the attack succeeded
Events

• Events are observable sequences of bytes that may be part of an attack (Flex and friends)

<table>
<thead>
<tr>
<th>Event</th>
<th>Token</th>
<th>Lexeme</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLOGIN</td>
<td>L</td>
<td>(^“230”(\w)\n)</td>
<td>User logged in</td>
</tr>
<tr>
<td>QUIT</td>
<td>Q</td>
<td>(^“QUIT”\n)</td>
<td>User Quit</td>
</tr>
<tr>
<td>CWD</td>
<td>C</td>
<td>(^“CWD”)</td>
<td>Change Directory</td>
</tr>
<tr>
<td>ARG</td>
<td>A</td>
<td>([SP] &lt;str&gt; \n)</td>
<td>Argument of an FTP command</td>
</tr>
<tr>
<td>INVALID</td>
<td>$l$</td>
<td>(^[^1-5])</td>
<td>A non-FTP response</td>
</tr>
</tbody>
</table>

• Protocol Dependent

• Libraries for standard protocols
Regular Expressions

- **Precondition** \(((\neg L)^* \cdot L \cdot (\neg Q)^*)^+\)

- **Exploitation**
  
  \[ C \cdot (A \text{ such that } data \in (.)^*\text{bin/sh(.)}* \&\& length>100) \]
Regular Expressions (cont.)

- Confirmation $I_R$

- Each expression defines a language $L_{\text{pre}}, L_{\text{exp}}, L_{\text{conf}}$
Putting the Signature Together

- GARD uses Hierarchical State Machines
Invariant Specification

- Invariant is a logical formula over the state variables of the finite state machine.

<table>
<thead>
<tr>
<th>Var.</th>
<th>Values</th>
<th>Semantic Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>{0, 1}</td>
<td>A USER command was issued.</td>
</tr>
<tr>
<td>x2</td>
<td>{0, 1}</td>
<td>A PASS command was issued.</td>
</tr>
<tr>
<td>x3</td>
<td>{0, 1}</td>
<td>Victim has indicated a successful login.</td>
</tr>
<tr>
<td>x4</td>
<td>{U = 0, A = 0, B =1, E=2}</td>
<td>Holds session representation type</td>
</tr>
<tr>
<td>x5</td>
<td>{U = 0, S = 0, B =1, C=2}</td>
<td>Holds session transmission mode</td>
</tr>
<tr>
<td>x6</td>
<td>{0, 1}</td>
<td>A session is in passive mode.</td>
</tr>
<tr>
<td>x7</td>
<td>{0, ... ,MAX}</td>
<td>Number of files uploaded in this session.</td>
</tr>
<tr>
<td>x8</td>
<td>{0, ... ,MAX}</td>
<td>Number of files downloaded in this session.</td>
</tr>
</tbody>
</table>
### Events and Variables

<table>
<thead>
<tr>
<th>Event</th>
<th>Token</th>
<th>Lexeme</th>
<th>Pre-condition</th>
<th>Post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLOGIN</td>
<td>L</td>
<td>(^“230”\w\n)</td>
<td>-</td>
<td>x3=1</td>
</tr>
<tr>
<td>QUIT</td>
<td>Q</td>
<td>(^ “QUIT”\n)</td>
<td>-</td>
<td>∀ xᵢ = 0</td>
</tr>
<tr>
<td>CWD</td>
<td>C</td>
<td>(^“CWD”\n)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ARG</td>
<td>A</td>
<td>([SP] &lt;str&gt; \n)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>INVALID</td>
<td>Iᵣ</td>
<td>(^[^1-5])</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

- We can translate the logical formula to a regular language, L(Iⁿ)
The whole picture

A,E,L
x1=0,... xn=0
L
Q
x1=1,... xn=0
A,L
A,E,L
x1=1,... xn=0
A,C,E,Q
x1=1,... xn=0
Q
C
Q
x1=1,... xn=0
Super State
Signature Evaluation

• Define
  
  - \( L(\text{SS}) = L_{\text{pre}} \cdot L_{\text{exp}} \cdot L_{\text{conf}} \)
  
  - \( L(I_{\text{ftp}}) \)
  
  - \( U_{\text{FTP}} = \text{ultimate set of attacks} \)

• Ideally we would like \( L(\text{SS}) = U_{\text{FTP}} \)

• Non-ideal situation generates false positives and false negatives.
  
  - \( fp = L(\text{SS}) \cap \neg U_{\text{FTP}} \), \( fn = \neg L(\text{SS}) \cap U_{\text{FTP}} \)
Signature Evaluation (cont.)

- The methodology assumes $L(I_{ftp}) \supseteq U_{FTP}$

- But now we have to deal with spurious ($sp$) sequences.
Edit Distance

- Systematic method requires an iterative refinement

- Reduce the probability of sp, generate new instances through modifications to existing instances
  - Edit distance: \( ed(s_1, s_2) = \) number of deletions, insertions or substitutions to transform \( s_1 \) to \( s_2 \)
  - \( ED_k(L) = \{ x | \exists y \in L \text{ such that } ed(x, y) < k \} \).
Modeling the Protocol

- Given a protocol $P$, we construct a semantic model of $M_P$ (a finite state machine).
- A state in $M_P$ is a valuation of variables, transitions affect these variables.
Some pitfalls

• Operations on languages introduce $fp$ or $fn$.
  – Union introduces extra paths
    • Not really an attack
    • An attack not captured by the session signature.

• GARD guarantees no false positives and no false negatives with respect to the invariant

• Domain experts come up with both the invariant and the session signatures
  – GARD assists in narrowing down $fp$ and $fn$ through automatic generation of attacks.
Automatic Generation and Analysis of NIDS Attacks

- Edit distance is one approach
- Attackers can be (and usually are) sneaky
  - Split the attack into multiple FTP sessions
    1. Login and ftp over code and log out
    2. Login and execute code from (1)

Problem

- Given an attack instance automatically generate all possible instances
- Verify that these are attacks!
The problem(s) ...

- **Black Hat Problem**
  - Given an NIDS and an instance of an attack \( \mathcal{A} \), find an instance of \( \mathcal{A} \) that evades the NIDS

- **White Hat Problem**
  - Given an instance of an attack \( \mathcal{A} \) and a sequence of packets \( s \), determine whether \( s \) is an instance of \( \mathcal{A} \)
How do they do it?

• An attacker knows
  – The signature(s) used
  – The protocol(s) e.g., ftp, TCP etc.
  – An instance of the attack

• Based on the above knowledge
  – Perform transformations/rewrites on one attack instance to obtain a new attack instance
We'll do the same ...

- Attacker's knowledge as inference (or transformation) rules
- Use an inference engine to generate all possible attack instances
  - Starting from a known attack instance
- White Hat Problem: run the inference engine
- Black Hat Problem: check if the attack is a member of the set returned by the inference engine
Limitations

• **Black Hat - Infinite traces**
  – Partitions based on testing techniques
    • Each partition exercises different features an NIDS should handle
  – Prune some derivations
    • No packet fragmentation on packets with size less than 5 bytes

• **White Hat – when to stop searching**
  – Bottom up approach (shrinking rules)
Rules

• Application, Protocol Rules, OS Rules
• Split into two categories
  – Shrinking Rules
  – Expanding Rules
• TCP Fragmentation (r1)
  – Fragments an attack packet into two packets. Adds victim acknowledgment after each new packet.
• HTTP space padding (r7)
  – Insert spaces after an HTTP method: from “GET <URL>” into “GET___<URL>”
Formal Model of Attack Derivation

• Natural deduction system \(<F, \Phi>\)
  - \(F\) is the set of facts
  - \(\Phi\) is the set of inference rules

• Derivations
  - \(f1 \vdash^\phi fn\), if there is a derivation sequence \(<f1, ..., fn>\) such that \(f1 \in F\) and each \(f_{i+1}\) is a result of applying a derivation rule \(r \in \Phi\).
Assumptions

- Each rule has an expanding and a shrinking version.
- A derivation containing only shrinking rules has not cycles.
- Root(a)
  - A derivation containing only shrinking rules and starts from sequence a
Derivation model of an attack

- Given $\alpha$ as an instance of an attack $\mathcal{A}$ and a set of inference rules $\Phi$
  - A derivation model of $\mathcal{A}$ is a natural deduction system of $<\text{roots}_\Phi(\alpha), \Phi>$
  - The closure of a derivation model ($\text{Cl}_\Phi(\text{roots}_\Phi(\alpha))$) is the set of all TCP sequences that are derived from $\text{roots}_\Phi(\alpha)$ using $\Phi$'s rules.
Black Hat and White Hat

• NIDS view
  - N is a NIDS, N's view with respect to an attack $\mathcal{A}$ is the set of TCP sequences that N recognizes as $\mathcal{A}$

• Black Hat
  - Given $<\text{roots}_\phi(\alpha), \Phi>$ for $\mathcal{A}$, and an NIDS view of $\mathcal{A}$ denoted as $V_{\mathcal{N}\mathcal{A}}$, find $s \in \text{Cl}_\phi(\text{roots}_\phi(\alpha)) \setminus V_{\mathcal{N}\mathcal{A}}$

• White Hat
  - Given $<\text{roots}_\phi(\alpha), \Phi>$ for $\mathcal{A}$, find $s \in \text{Cl}_\phi(\text{roots}_\phi(\alpha))$
Properties of the Attack Derivation Model

- For an attack $\mathcal{A}$ and a set of rules $\Phi$ a derivation model is
  - Sound if it derives TCP sequences that implement $\mathcal{A}$,
  - Complete if it can derive any TCP sequences that implements $\mathcal{A}$
  - Decidable given a TCP sequence there is an algorithm that determines whether or not a sequence is derived from the root.
For our two Hat Problems

• Black Hat
  – Soundness
  • Any instance we discover is a vulnerability
  – Completeness
  • Eventually the model will generate all instances

• White Hat
  – Soundness
  • Lack of false positives
  – Completeness
  • Lack of false negatives
Proving Completeness

• There is no formal definition of the notion
  – “a TCP sequence that implements $A$”

• However, the derivation model can be used to inductively define “implements” $A$.
  – Each transformation rule preserves $A$'s semantics.