Prospex: Protocol Specification Extraction

Presented by: Yiyang Shao, NSLab, Tsinghua University

Original Authors:
Paolo Milani Comparetti paolo@iseclab.org, Vienna University of Technology
Gilbert Wondracek gilbert@iseclab.org, Vienna University of Technology
Christopher Kruegel chris@cs.ucsb.edu, UC Santa Barbara
Engin Kirda engin.kirda@eurecom.fr, Institute Eurecom, Sophia Antipolis
Motivation

• Stateful protocol specifications can be used for
  – Blackbox vulnerability analysis
  – Automated fuzz testing
  – Deep packet inspection
  – Intrusion detection
  – Show differences between implementations of protocols
    • Fingerprinting
    • Testing
Motivation

• Manual network protocol reverse engineering is a timeconsuming and tedious task
• Goal: Automatic extraction of application level protocol specifications
• Several systems exist that can automatically extract precise message formats for individual messages, however they do not aim at extracting a protocol state machine
• Prospex aims at producing detailed specifications for stateful protocols
Our Contributions

• We present
  – a technique for automatically determining message types
  – a novel way for inferring a minimal automaton that is consistent with a set of application sessions (state machine)

• Our system is the first to automatically infer specifications for stateful protocols

• Specifications for fuzz testing are automatically generated from the recovered specifications
System Overview

- Our system operates in four phases
- Each phase produces input for the following phase
Session Analysis Phase

- How is the server processing messages?
  - Behavior based approach

- Record an execution trace
  - Run the application (server) in a dynamic data tainting environment
  - Assign a label to each input byte, track its propagation during the execution
  - Do this while engaging the server in a series of application sessions (using a client)
  - For example, observe sendmail (SMTP daemon) while using a mail client to send mail
  - Yields an execution trace for a session, containing all executed instruction and taint labels of all instruction operands
Message Format Inference

- Apply a set of techniques and heuristics to the execution trace
- Described in previous work (Automatic Network Protocol Analysis, NDSS 2008)
- Allows us to recover message formats for individual messages
- Each message format is represented as a tree of nested fields
Message Clustering Phase

Dynamic taint analysis

Input data

Application

Controlled environment

Session analysis

Message format inference

Execution trace analysis

Message clustering

Feature extraction

Clustering

State machine inference

State machine minimization

State labeling

Clusters
Message Clustering

- After the session analysis phase, we have format specifications for individual messages.
- We want to automatically determine the different message types that appear in the observed application sessions.
- Assume a similar “reaction” of the server to similar messages (e.g. if they have the same type).
- First step: Find a metric of similarity between messages.
Message Similarity

- We define several similarity features and distance functions
- These features can be divided into three groups:
  - Input similarity features ("message format")
  - Execution similarity features ("code execution")
  - Impact similarity features ("behavior")
- For each group, we compute a similarity score
Input Similarity Feature

• Assumption: Messages of the same type have a similar field structure
• To compute an input similarity score, we use a modified sequence alignment algorithm (hierarchical Needleman – Wunsch)
• The sequences of fields for all message formats are compared
• Similar parts get aligned, exposing differences or missing parts (matches, mismatches, gaps)
Execution Similarity Features

- Assumption: Similar messages are handled by similar code
- For each pair of messages, the sets of
  - system calls
  - process activity (clone, kill,...)
  - invoked functions
  - invoked library functions
  - executed addresses
  are recorded
- Then, the Jaccard indices (measure of set similarity) are computed:

\[ J(a, b) = \frac{|a \cap b|}{|a \cup b|} \]
Impact Similarity Features

- Assumption: Similar messages trigger similar behavior in the server application
- Output similarity feature
  - Captures the output behavior of the server, based on destination and taint status
  - Four possible destinations considered:
    - Client’s socket, other socket, files, terminal
  - Taint status
    - Previously tainted (e.g. echoed) or not
  - For each message, as a sequence of tuples \(<\text{sink}, \text{taint}>\) is considered (consecutive duplicates removed)
  - Needleman Wunsch is used to compute the output similarity score
Impact Similarity Features

- File system feature
  - Captures the server file system activity
    - We consider system calls that perform FS actions like opening a file, getting info on a directory, etc.
  - Sets of <operation, path> tuples are assigned to each message
  - “path” needs to be generalized
  - For each part of the path, we check if it is
    - Hardcoded in the binary
    - Tainted (“TAINT”)
    - Contained in an (optionally provided) config file (“CONFIG”)
    - Neither tainted nor in config file (“VARIABLE”)
  - Examples: <open, “/CONFIG/TAINT”>, <write, “/var/log/samba/VARIABLE”>
  - The similarity distance is then computed using the Jaccard index
Clustering

- The similarity features are used to compute a distance matrix
  \[ d(a,b) = 1 - \sum_i \omega_i s_i(a,b) \]
- We apply the partitioning around medoids (PAM) algorithm for clustering
- PAM needs the desired number of clusters \( k \) as a parameter
- We determine \( k \) by employing a generalization of the Dunn index
  - Dunn index is a standard intrinsic measure of clustering quality (cluster separation / cluster compactness)
- Result: Clusters of messages that are similar (e.g. same type)
- For each cluster, a generalized message format is generated
State Machine Inference
State Machine Inference

• Goal: Use the information on message types and the application sessions that we observed to infer a minimal state machine
• Find the minimal automaton that is consistent with our training set, without being overly general
• We start by constructing an Augmented Prefix Tree Acceptor (APTA)
  • APTA = Incompletely specified DFA with a state transition graph that is a tree
  • Each branch of the tree represents the sequence of message types within an observed application session
Augmented Prefix Tree Acceptor (APTA)

- Agobot (malware) example with 2 application sessions in the training set:

- We want to generalize the APTA by merging some states
- We only want to merge states that correspond to similar states in the server application (otherwise overly general)
Minimization

- Goal: Identify and merge similar states
- Commonly, in application level protocols, specific messages have to be sent before the server can perform certain actions
  - For example, often a login is necessary before other commands can be executed
  - Other commands may lead the server away from these states
- Identify states where the application can process similar commands based on the sequence of messages that it previously received
State Labeling

- To capture this intuition, we use prerequisites.
- A prerequisite is a sequence of messages that the server has received that leads it to a specific state.
- For the server to be in a state where it can meaningfully process a message of type $m$, it first has to receive a message of type $r$ (always), optionally followed only by messages of certain types.
- Algorithm to find all prerequisites presented in paper.
- Once all prerequisites are computed, each state is labeled with the set of message types that are allowed as input in that state.
- $m$ is allowed in a state if the sequence of message types leading to this state matches all prerequisites for $m$. 
Labeled Example
State Machine Minimization

• Compute the minimal consistent DFA from the labeled APTA to get the state machine:
  – Endstate detection
    • Simple heuristic: Mark endstates by finding messages that only appear last in sessions
  – Apply a known algorithm:
    • Exbar is the state-of-the-art exact algorithm for minimal consistent DFA inference
    • Prospex runs Exbar on the labeled state tree
  – Result is the protocol state machine
Agobot Example

• Example state machine (generated from 2 observed application sessions):

• Captures the notion that “login “ is necessary for the command, and “logout “ returns to the initial state
Evaluation

• We created state machines for 4 widely deployed realworld protocols

• Agobot
  – Malware example
  – Textbased protocol (close to IRC), bots often use custom C&C protocols
  – We mimicked a bot herder and performed a few commands on our own IRC server

• SMTP
  – Applied our system to sendmail daemon
  – Used 16 application sessions (sending email) as training set
Evaluation

- **Server Message Block (SMB)**
  - Complex, stateful, binary protocol
  - We observed the smbd daemon
  - Used smbclient for creating the training set
  - Recorded 31 training sessions, performing file operations

- **Session Initiation Protocol (SIP)**
  - Textbased protocol
  - Often used in VOIP infrastructure
  - Asterisk server
  - Connected using 2 client softphones, made phone calls, using voice boxes etc.
  - Training set represents the use cases of calling someone
Example State Machines

**SMTP**

**SIP**
Example State Machines

SMB
Quality of Specifications

- How good are these results?
  - Specifications should parse valid sessions without being too general

- Parsing success
  - “Complete” means not overly restrictive, e.g. the inferred state machine parses valid sessions
  - To this end, we parsed real-world network traces with the extracted specifications

<table>
<thead>
<tr>
<th>Protocol</th>
<th>#Sessions</th>
<th>Parsing success</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMTP</td>
<td>31,903</td>
<td>93.5%</td>
</tr>
<tr>
<td>SIP</td>
<td>80</td>
<td>100%</td>
</tr>
<tr>
<td>SMB</td>
<td>80</td>
<td>90%</td>
</tr>
</tbody>
</table>

- SMTP: remaining 6.5% used TLS encryption (limitation)
- SMB: Fails were previously unknown error conditions (files not found etc.)
Quality of Specifications

- State machine comparison:
  - We built reference state machines for the tested protocols
  - Performed 50,000 random walks over inferred state machines, and checked if the message sequences are valid in the reference state machines
  - Precision: Ratio of sequences generated by random walks over the inferred state machine that are accepted by the reference state machine
  - Recall: Ratio for sequences from random walks over the reference state machine that are accepted by the inferred state machine

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agobot</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SMTP</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SMB</td>
<td>1</td>
<td>.58</td>
</tr>
<tr>
<td>SIP</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Comparative Evaluation

- Compare our state machine inference with known algorithms for inductive inference (based on Minimum Message Length)
  - Sk-strings, beams
- Known algorithms did not provide acceptable performance on our training data

<table>
<thead>
<tr>
<th></th>
<th>Agobot P</th>
<th>Agobot R</th>
<th>SMTP P</th>
<th>SMTP R</th>
<th>SMB P</th>
<th>SMB R</th>
<th>SIP P</th>
<th>SIP R</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prospex beams</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>.58</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>skstrings(and)</td>
<td>.56</td>
<td>1</td>
<td>.89</td>
<td>1</td>
<td>1</td>
<td>.50</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>skstrings(or)</td>
<td>.79</td>
<td>.20</td>
<td>1</td>
<td>.88</td>
<td>1</td>
<td>.30</td>
<td>1</td>
<td>.01</td>
</tr>
<tr>
<td>.11</td>
<td>.92</td>
<td>.11</td>
<td>.11</td>
<td>1</td>
<td>.12</td>
<td>.62</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Application: Fuzz testing

• Prospex can create fuzzing specifications from the extracted message formats and the state machine
• We contributed to the open source Peach Fuzzing Platform (statefulness) and applied the system to two applications

• SMB
  – 2,100 lines of Peach XML created
  – Found a file traversal vulnerability in smbd that allows downloading of /etc/passwd (filename semantic)

• SIP
  – Found a bug that segfaults Asterisk when a return value is set to “0”

• Vulnerabilities were unfortunately already known
• Nonstateful fuzzing would not get to these vulnerabilities
Limitations

• We limit ourselves to the analysis of the communication in a single direction, but both communication partners could be monitored simultaneously, combining the state machines and message formats
• We cannot handle encrypted network traffic
• Quality of specifications is limited by quality and variety of training data, e.g. observed sessions
Conclusion

- Prospex can automatically infer protocol specifications for stateful protocols
- Automatically identify message types
- Infer the protocol state machine
- Generate protocol specifications for a stateful fuzzer
Thanks for your attention!