MetaSymploit: Day-One Defense against Script-based Attacks with Security-Enhanced Symbolic Analysis

Ruowen Wang, Peng Ning, Tao Xie, and Quan Chen, *North Carolina State University*

This paper is included in the Proceedings of the
22nd USENIX Security Symposium.

August 14–16, 2013 • Washington, D.C., USA

ISBN 978-1-931971-03-4

Open access to the Proceedings of the
22nd USENIX Security Symposium
is sponsored by USENIX

MetaSymploit: Day-One Defense Against Script-based Attacks with Security-Enhanced Symbolic Analysis

Ruowen Wang, Peng Ning, Tao Xie, Quan Chen
Department of Computer Science
North Carolina State University
Raleigh, NC, USA
{*rwang9, pning, qchen10*}*@ncsu.edu, xie@csc.ncsu.edu*

Abstract

A script-based attack framework is a new type of cyber-attack tool written in scripting languages. It carries various attack scripts targeting vulnerabilities across different systems. It also supports fast development of new attack scripts that can even exploit zero-day vulnerabilities. Such mechanisms pose a big challenge to the defense side since traditional malware analysis cannot catch up with the emerging speed of new attack scripts. In this paper, we propose MetaSymploit, the first system of fast attack script analysis and automatic signature generation for a network Intrusion Detection System (IDS). As soon as a new attack script is developed and distributed, MetaSymploit uses security-enhanced symbolic execution to quickly analyze the script and automatically generate specific IDS signatures to defend against all possible attacks launched by this new script from Day One. We implement a prototype of MetaSymploit targeting Metasploit, the most popular penetration framework. In the experiments on 45 real attack scripts, MetaSymploit automatically generates Snort IDS rules as signatures that effectively detect the attacks launched by the 45 scripts. Furthermore, the results show that MetaSymploit substantially complements and improves existing Snort rules that are manually written by the official Snort team.

1 Introduction

Over the years, with rapid evolution of attacking techniques, script-based attack frameworks have emerged and become a new threat [2, 3, 6, 39]. A script-based attack framework is an attack-launching platform written in scripting languages, such as Ruby and Python. Such framework carries various attack scripts, each of which exploits one or more vulnerabilities of a specific application across multiple versions. With the high productivity of using scripting languages, attackers can easily develop new attack scripts to exploit new vulnerabilities.

To launch an attack, an attacker runs an attack script on the framework remotely. By probing a vulnerable target over the network, the attack script dynamically composes an attack payload, and sends the payload to the target to exploit the vulnerability. The attack framework also provides many built-in components with APIs of various attack functionalities to support rapid development of new attack scripts. Once a zero-day vulnerability is found, a new attack script can be quickly developed and distributed in hacking communities, where other attackers even script kiddies can directly download the new script to launch attacks exploiting the zero-day vulnerability.

A well-known example of the script-based attack frameworks is Metasploit [3], the most popular Ruby-based penetration framework. It has more than 700 attack scripts targeting various vulnerable applications on different operating systems (OSes). It also provides built-in components for creating new attack scripts. Metasploit was originally developed for penetration testing using proof-of-concept scripts. But with years of improvements, it has become a full-fledged attack framework. Unfortunately, as an open source project, Metasploit can be easily obtained and used by attackers for illegal purposes. For example, it was reported that the well-known worm "Conficker" used a payload generated by Metasploit to spread [5]. A Metasploit attack script was immediately distributed after a zero-day vulnerability was found in Java 7 [32]. A four-year empirical study shows real malicious network traffic related to Metasploit on a worldwide scale. Moreover, the study shows that many Metasploit attack scripts are used by attackers almost immediately after the scripts are distributed in hacking communities [33].

When a new attack script is distributed and captured by security vendors, the traditional approach to defend against it is to first set up a controlled environment with a vulnerable application installed. Then security analysts repeatedly run the script to exploit the environment over a monitored network, collecting a large number of at-

tack payload samples, and finally extract common patterns from the samples to generate IDS signatures.

However, with the attack framework, new attack scripts can be quickly developed and distributed to exploit the latest vulnerabilities. This poses a great challenge that the traditional approach can hardly catch up with the release speed of new attacks, due to the time-consuming process of setting up test environments and analyzing attack payload samples. In our evaluation (Section 5), we observe that even the latest Snort IDS rules written by security analysts cannot detect many Metasploit-based attacks.

In this paper, we propose *MetaSymploit*, the first system of fast attack script analysis and automatic IDS signature generation. As soon as a new attack script is distributed, MetaSymploit quickly analyzes the attack script and automatically generates IDS signatures of its attack payloads, thereby providing defense against new attacks launched by this script from Day One. Particularly, MetaSymploit gives the *first aid* to zero-day vulnerabilities whose security patches are not available while the attack scripts that exploit them are already distributed.

Specifically, MetaSymploit leverages symbolic execution while enhancing it with several security features designed for attack script analysis and signature generation. By treating environment-dependent values as symbolic values, MetaSymploit symbolically executes attack scripts without interacting with actual environments or vulnerable applications, thus substantially reducing the time and cost of the analysis. With path exploration of symbolic execution, MetaSymploit also explores different execution paths in an attack script, exposing different attack behaviors and payloads that the script produces under different attack conditions.

To generate signatures of attack payloads, instead of analyzing large volumes of payload samples, MetaSymploit keeps track of the payload composing process in the attack script during symbolic execution. MetaSymploit uses symbolic values to represent variant contents in a payload (e.g., random paddings), in order to distinguish constant contents (e.g., vulnerability-trigger bytes) from variant ones. When the script sends a composed payload to launch an attack, MetaSymploit captures the payload's entire contents, extracts constant contents as patterns and generates a signature specific to this payload.

In a case study, we implement a security-enhanced symbolic execution engine for Ruby, develop MetaSymploit as a practical tool targeting Metasploit, and generate Snort rules as IDS signatures. Particularly, instead of heavily modifying the script interpreters, we design a lightweight symbolic execution engine running on unmodified interpreters. This lightweight design can keep pace with the continuous upgrades of the language syntax and interpreter (e.g., Ruby 1.8/1.9/2.0). Therefore, our design supports analyzing attack scripts written in different versions of the scripting language.

We evaluate MetaSymploit using real-world attack scripts. We assess our automatically generated Snort rules by launching attacks using 45 real-world Metasploit attack scripts from `exploit-db.com`, including one that exploits a zero-day vulnerability in Java 7. Our rules successfully detect the attack payloads launched by the 45 scripts. Furthermore, we also compare our rules with the official Snort rule set written by security analysts, and have three findings: (1) the official rule set is incomplete and 23 of the 45 attack scripts are not covered by the official rule set; (2) for the scripts covered by the official rules, our rules share similar but more specific patterns with the official ones; (3) our studies also expose 3 deficient official rules that fail to detect Metasploit attacks. Therefore, MetaSymploit is a helpful complement to improve the completeness and accuracy of existing IDS signatures to defend against attack scripts.

In summary, we make three major contributions:

1. We point out the security issues of script-based attacks, and propose a scalable approach called MetaSymploit that uses security-enhanced symbolic execution to automatically analyze attack scripts and generate IDS signatures for defense.

2. We implement a security-enhanced symbolic execution engine for Ruby and develop a practical tool for the popular Metasploit attack framework. Our tool can generate Snort rules to defend against newly distributed Metasploit attack scripts from Day One.

3. We demonstrate the effectiveness of MetaSymploit using recent Metasploit attack scripts in real-world attack environments, and also show that MetaSymploit can complement and improve existing manually-written IDS signatures.

2 Background

We first give the background of how an attack script works. Generally, when an attack script runs on top of an attack framework, the script performs four major steps to launch an attack. (1) The script probes the version and runtime environment of the vulnerable target over the network. (2) Based on the probing result and the script's own hard-coded knowledge base, the script identifies the specific vulnerability existing in this target. The knowledge base is usually a list containing the information (e.g., vulnerable return addresses) of all targets that this script can attack. (3) Then the script dynamically composes an attack payload customized for this target. (4) Finally, the script sends the payload to the target to exploit the vulnerability.

```
1  def exploit
2    connect()
3    preamble = "\x00\x4d\x00\x03\x00\x01"
4    version = probe_ver()
5    if version == 5
6      payload = prep_ark5()
7    else
8      payload = prep_ark4()
9    end
10   preamble << payload.length
11   sock.put(preamble) # Required by protocol
12   sock.get_once()
13   sock.put(payload) # Send attack payload
14   sock.get_once()
15   ... # vulnerability triggered
16 end
17 def prep_ark5()
18   payload = shellcode()
19   payload << rand_alpha(1167 -
         payload.length)
20   payload << "\xe98" + [-1172].pack("V")
21   payload << "\xeb\xf9"
22   payload << get_target_ret(5) # Tar_Ver: 5
23   payload << rand_alpha(4096 -
         payload.length)
24   return payload
25 end
```

Listing 1: The code snippet from a real Metasploit attack script *type77.rb* [4] (slightly modified for better presentation)

Depending on the attack strategy and vulnerability type, different scripts may have different attack behaviors when performing these steps. For example, a brute-force attack may keep composing and sending payloads with guessed values until the target is compromised, while a stealthy attack may carefully clean up the trace in the target's log after sending the payload.

Among these steps, composing and sending an attack payload are the key steps of launching an attack. An attack payload is typically a string of bytes composed with four elements: (a) special and fixed bytes that can exploit a specific vulnerability; (b) an arbitrary shellcode that attackers choose to execute after the vulnerability is exploited. The shellcode content is usually variant, especially when obfuscated; (c) random or special paddings (e.g., NOP 0x90) that make the payload more robust; (d) other format bytes required by network protocols.

With the help of the rich libraries of scripting languages and the built-in components provided by the attack framework, an attack script can call APIs of related libraries or components to help it perform each step, especially composing an attack payload.

As an example, Listing 1 shows a Ruby code snippet extracted from a real Metasploit attack script exploiting a vulnerable application called Arkeia. In the example, the script defines two methods. `exploit` is the main method that performs the major steps to launch the attack. `prep_ark5` is one of the payload composing methods. When the script runs on Metasploit, it first

```
alert tcp any any -> any 617 (
msg:"Script: type77 (Win), Target Version: 5,
  Behavior: Version Probing, Stack Overflow,
  Pattern: JMP to Shellcode with
       Vul_Ret_Addr";
content:
"|e9 38 6c fb ff ff eb f9 ad 32 aa 71|";
pcre:"/[.]{1167}\xe9\x38\x6c\xfb\xff\xff\xeb\
    xf9\xad\x32\xaa\x71[a-zA-Z]{2917}/";
classtype:shellcode-detect; sid:5000656;)
```

Listing 2: One Snort rule signature generated for the attack payload composed by `prep_ark5`.

connects to the target over the network (Line 2), and then probes the target's version (Line 4). Here both `connect` and `probe_ver` are API methods of a built-in network protocol component. Based on the version, it calls the corresponding method to start composing the attack payload specific to the target (Lines 5-9).

When `prep_ark5` is called, the payload is first assigned by the shellcode component, which returns a configured shellcode (Line 18). Note that the shellcode can be freely chosen and obfuscated. The shellcode component offers several different shellcodes for different purposes. Then the payload is appended (<<) with several contents (Lines 19-23). `rand_alpha` generates random alphabet padding to not only extend the payload to the required size of the network protocol, but also introduce more randomness for evasion. The concrete bytes represent some assembly code that will jump to the shellcode (e.g., "\xeb\xf9" and "\xe9" are two JMP instructions). `pack("V")` converts the integer to bytes as the offset of one JMP. `get_target_ret` is another attack framework API that queries the script's knowledge base (omitted here due to space limit, please refer to [4]) to retrieve the exploitable return address based on the target version, which can hijack the control flow[1] (Line 22). After the payload is composed, the script first sends a preamble packet to the target, followed by the attack payload packet to exploit the vulnerability (Lines 11-13).

Popular attack frameworks provide plenty of built-in components covering various network protocols, OSes, and offering different shellcodes and NOP paddings, which enable attackers to quickly develop new attack scripts to exploit different targets. Furthermore, advanced attackers can create even sophisticated attack scripts, which have multiple execution paths performing different attack behaviors and payloads. Some of them may be triggered only under certain attack conditions.

Therefore, the traditional approach that requires both controlled environments and vulnerable applications is not scalable for analyzing attack scripts. Since differ-

[1] In [4], the exploitable return address actually points to a POP/POP/RET instruction sequence, which is a typical SEH-based attack to hijack control flow in Windows.

ent attack scripts target different applications and OSes, it is costly and time-consuming to obtain every application (let alone the expensive commercial ones) and set up environments for every OS. It is even harder to create different attack conditions to expose different attack behaviors and payloads in sophisticated attack scripts.

3 MetaSymploit

In this section, we first state the problem and assumptions we focus on, and then give an overview of MetaSymploit, followed by the detailed techniques in its two core parts.

3.1 Problem Statement and Assumptions

Problem Statement. We focus on the problem caused by script-based attack frameworks and their attack scripts: how to provide an automated mechanism that can analyze and defend against newly distributed attack scripts. Particularly, the mechanism should be time-efficient in order to address the security issues caused by two major features of attack scripts: a large number of scripts with wide-ranging targets, and fast development and distribution of new scripts that can be directly used to exploit zero-day vulnerabilities.

Assumptions. We assume that both script-based attack frameworks and attack scripts are available from either public or underground hacking communities. As soon as a new attack script is distributed, it can be immediately captured and analyzed. We also assume that the scripting languages used by attack frameworks are general-purpose object-oriented scripting languages, such as Ruby and Python. In reality, `sectools.org` lists 11 most popular attack tools [6] in the public community. 8 of them are Ruby/Python-based attack frameworks. Most of them are actively maintained with frequent updates of new attack scripts.

3.2 MetaSymploit Overview

Given an attack script, the goal of MetaSymploit is to quickly analyze fine-grained attack behaviors that the script can perform, and automatically generate specific IDS signatures for every attack payload that the script can compose, providing a fast and effective defense against attacks launched by this script. To achieve this goal, MetaSymploit leverages symbolic execution and enhances it with a number of security features designed for attack scripts analysis and signature generation.

Symbolic execution[2] is a program analysis technique that executes programs with symbolic rather than concrete values. When executing branches related to symbolic values, it maintains a path constraint set and forks to explore different execution paths. By using symbolic execution, MetaSymploit has three advantages to achieve fast analysis and defense against attack scripts: (1) analyzing scripts without requiring actual environments or vulnerable targets, (2) exploring different execution paths to expose different attack behaviors, (3) using symbolic values to represent variant contents in attack payloads to ease the extraction of constant patterns.

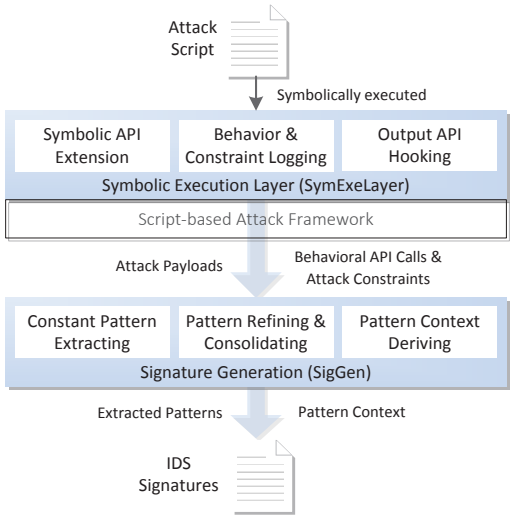

Figure 1: MetaSymploit consists of two major parts drawn in grey. (The arrows show the workflow of an attack script analysis.)

Figure 1 shows the architecture of MetaSymploit, which consists of two major parts, the symbolic execution layer (SymExeLayer) and the signature generator (SigGen). Given an attack framework, SymExeLayer is built upon the framework. It reuses the framework's execution facility while extending the framework interface to support symbolic execution of attack scripts. When a script is symbolically executed, SymExeLayer captures all attack behaviors and payloads that the script can perform and compose. After the symbolic execution is done, SigGen takes the captured results as inputs. It extracts constant patterns by parsing the contents of the attack payloads. It also analyzes the attack behaviors to derive the semantic contexts that describe the extracted patterns. Finally, SigGen combines the patterns and the contexts to generate IDS signatures for this attack script.

More specifically, three key techniques are developed to realize the functionalities of SymExeLayer and SigGen, respectively. As shown in Figure 1, SymExeLayer consists of (1) *Symbolic API Extension*. It extends the APIs of both the attack framework and the scripting language to support symbolic values and operations. Notably, it extends the APIs related to environments/tar-

[2]For more background of symbolic execution, please refer to [25]

gets and variant payload contents to return symbolic values. (2) *Behavioral API & Attack Constraint Logging*. It records critical API calls that represent attack behaviors. It also logs path constraints of symbolic values related to environments and targets. Both logs will be used for deriving pattern context (described later). (3) *Output API Hooking*. It hooks various output APIs that are used to send attack payloads, in order to capture complete payload contents for extracting constant patterns.

SigGen consists of (1) *Constant Pattern Extracting*. By parsing the payload contents, it extracts constant patterns that can represent the payload. Constant patterns include fixed contents, fixed lengths of contents, and fixed offsets of the contents in the format. (2) *Pattern Refining and Consolidating*. It refines patterns by distinguishing critical patterns from common benign bytes and trivial patterns. It also avoids generating duplicated signatures by examining repeated patterns. (3) *Pattern Context Deriving*. In order to describe what the extracted pattern represents, it analyzes the logs of behaviors and constraints to derive the semantic context of the pattern.

To illustrate the workflow of MetaSymploit, we revisit the script in Listing 1. First, SymExeLayer takes the script as input and symbolically executes it. The script calls a number of symbolic-extended APIs, including `probe_ver`, `shellcode` and `rand_alpha`. Instead of returning a concrete number, `probe_ver` assigns `version` a symbolic integer representing the target version. `shellcode` and `rand_alpha` return symbolic strings to represent all possible shellcodes and random paddings, respectively. Meanwhile, `probe_ver` indicates the probing behavior. SymExeLayer logs it as one attack behavior. SymExeLayer also logs the path constraint `version==5` since it indicates that the Line 6 branch is taken only under the attack condition that the target version is 5. In contrast, when symbolic execution forks to explore Line 8, SymExeLayer logs the negated constraint `version!=5`.

When executing `prep_ark5`, SymExeLayer logs `shellcode`, `rand_alpha`, and `get_target_ret`, since these APIs indicate a typical attack behavior of composing a stack overflow payload. Note that because `get_target_ret` is a call with a concrete argument, SymExeLayer uses the underlying framework to execute it normally to get the concrete return address value. On the other hand, SymExeLayer symbolically extends the << API to support appending symbolic strings. Finally, when the composed payload is sent, the hooked output API `sock.put` captures the complete payload contents.

SigGen then analyzes the payload contents and the behavior & constraint logs to generate signatures. Listing 2 shows one Snort rule generated by SigGen. The `content` is the byte pattern extracted from the constant bytes in the payload composed in Lines 20-22. The first 8 bytes are two JMP instructions and the last 4 bytes are the return address. The `pcre` is a regular expression matching the entire payload packet, including constant bytes and random paddings. `content` provides general fast matching, while `pcre` provides more precise matching. The `msg` shows the pattern context. The target version is derived from the `version==5` constraint. The behavior and the meaning of the patterns are derived from the logged behavioral API calls. The `msg` gives more insights that guide security analysts to use the signature to protect vulnerable application of specific version.

3.3 Symbolic Execution Layer

This section explains more details about the three techniques of SymExeLayer that extend the attack framework to perform symbolic execution and attack logging.

3.3.1 Symbolic API Extension

The key point of performing symbolic execution on attack scripts is to treat all variant values involved in the attack launching process as symbolic values, so that all possible attack variations can be covered. Since attack scripts use APIs to operate variant values, we extend the variant-related APIs of both the scripting language and the attack framework with symbolic support.

The variant-related APIs can be further divided into two categories: direct and indirect. Direct-variant-related APIs always return variant values. There are two major types in this category, (1) the APIs probing external environments/targets, (2) the APIs generating random payload contents. In both cases, we replace the original APIs with our symbolic-extended ones, which directly return symbolic values when called. As a result, the first type of APIs skips probing the actual environment/target, such as `probe_ver` in the example. Such skipping makes MetaSymploit scalable and efficient, since there is no need to prepare different environments or applications when analyzing different scripts. For the second type, as the payload content is a string of bytes, the APIs use symbolic values to represent any variant bytes, such as `shellcode` and `rand_alpha`. Hence, we can clearly distinguish concrete contents from symbolic contents in one payload. In addition, every symbolic value is assigned with a label showing what it represents based on its related API, such as `sym_ver`, `sym_shellcode`, and `sym_rand_alpha`. Note that SymExeLayer uses these labels to keep the semantics of the values, rather than relying on variable names, which can be freely decided by attackers.

Indirect-variant-related APIs return variant values only when their arguments are variant values. Such case typically happens in the operations of some primitive

classes such as String, Integer, and some payload composing operations. In SymExeLayer, we extend such APIs by adding the logic of handling symbolic arguments. If the arguments are concrete, the APIs execute the original logic and return concrete values as normal. If the arguments are symbolic, the APIs switch to the symbolic handling logic, which propagates the symbolic argument in accord with the API functionality, and returns a symbolic expression. In Listing 1, for a concrete string argument, the symbolic-extended << appends it as normal. For a symbolic argument, it holds both the original string and the new appended symbolic one in order and returns them as one symbolic string expression.

3.3.2 Behavioral API & Attack Constraint Logging

Since symbolic execution is a general program analysis technique, in order to provide additional security analysis of attack scripts, for every execution path, we keep a log recording both critical API calls that reflect attack behaviors and path constraints that represent the attack condition when exploring each execution path.

Behavioral API Logging. As mentioned in Section 2, attack scripts use APIs provided by the language library and the attack framework to launch attacks. In the analysis, it is critical to capture the API calls that perform the detailed attack behaviors during the launching process. There are two major types of behavioral APIs, network protocol APIs and payload-related APIs. By logging the first type, we are able to capture all the interactions between the attack script and the target. By logging the second type, we know exactly how a payload is composed and keep track of its detailed format and contents.

In practice, given an attack framework, we build a knowledge base collecting the APIs from the libraries and components that provide network protocols and payload-related operations. During execution, SymExeLayer identifies behavioral APIs and logs them while keeping the API call sequence in the execution path. Note that we also log the arguments and return values of the APIs, especially for payload-related APIs, whose return values may be a part of the payload contents.

Attack Constraint Logging. In symbolic execution, path constraints are the set of branch conditions involving symbolic values in one execution path. When encountering a new symbolic branch condition, symbolic execution consults a constraint solver to decide which branch(es) is feasible to take, and adds the new branch constraint into the path constraint set. If both branches are feasible to take, the execution path `forks` into two paths to explore both branches [25].

In attack scripts, we focus on the constraints related to environments and targets. We regard these constraints as *attack constraints* because different symbolic conditions that they represent typically indicate different attack conditions reflecting the probing results of environments or targets, therefore leading to different execution paths that compose different payloads in consequence. In the example, `version==5 ? prep_ark5 : prep_ark4`.

Recall that the APIs that probe external environments and targets are symbolic-extended. The symbolic return values of these APIs carry the labels showing what external source they represent. When executing a symbolic branch condition, we check if any symbolic value with external-source label is involved. If so, we log the corresponding constraint. In the example, when `version==5` is executed, we find that `sym_ver` is an external source, and thus log the constraint.

In summary, this behavior & constraint logging provides a fine-grained analysis report that saves the time-consuming work for security analysts. More importantly, the behaviors and constraints logged in each execution path can be further parsed to derive the semantic context for the extracted patterns (discussed in Section 3.4.3).

3.3.3 Output API Hooking

After an attack script finishes composing an attack payload, the script sends the payload as a network packet to the target to exploit the vulnerability. This payload sending step is the exact point of launching an attack. In order to capture the complete content of the attack payload for pattern extraction, we hook the output APIs that are used by attack scripts for sending payload.

Starting from the network layer to the application layer in the OSI model, we keep a list of the output APIs and their corresponding network protocols from both the scripting language's own network library and the built-in components of the attack framework.

We symbolically extend the output APIs by overriding their functionality from sending real network packets to dumping the entire packets locally. By doing so, the entire network flow sent from the attack script can be dumped throughout the execution. To keep the semantic context of each dumped packet, we associate them with the behavior & constraint log of that execution path, so that later the payload packets can be identified and the extracted patterns can be correlated with the context derived from the log. In the example script, the hooked `sock.put` dumps two packets. With the associated log, we identify the payload packet for pattern extraction.

Note that as a part of the network protocol APIs, the output APIs are also behavioral APIs that need to be logged. In addition, we also include the corresponding network protocols in the log. Later during signature generation, the log gives a clear view of which network protocol is used, and therefore SigGen can apply the correct packet format when parsing the packet contents.

```
18: payload=>[<sym_shellcode, len=Sym_Int>]
19: payload=>[<sym_shellcode, len=Sym_Int>,
  <sym_rand_alpha, len=(1167-Sym_Int)>]
20-22: # Appending concrete substrings
payload => [<sym_shellcode, len=Sym_Int>,
  <sym_rand_alpha, len=(1167-Sym_Int)>,
  <"\xe9\x38\x6c\xfb\xff\xff\xeb\xf9
  \xad\x32\xaa\x71", 12>]
23: payload => [<sym_shellcode, len=Sym_Int>,
  <sym_rand_alpha, len=(1167-Sym_Int)>,
  <"\xe9\x38\x6c\xfb\xff\xff\xeb\xf9
  \xad\x32\xaa\x71", 12>,
  <sym_rand_alpha, 2917>]
```

Listing 3: The symbolic string form showing the content of `payload` when `prep_ark5` is executed. `Sym_Int` is a symbolic integer representing the size of the shellcode.

3.4 Signature Generator

Given the dumped payload packets and the logs as inputs, SigGen includes three techniques to generate signatures.

3.4.1 Constant Pattern Extracting

In order to generate a signature that can detect a payload packet, it is necessary to extract a set of constant patterns that always stay the same across different variations of the payload. Specifically, there are three constant patterns that can be extracted: fixed-content pattern, fixed-length pattern and fixed-offset pattern. For ease of explanation, we first present the formal form of a dumped symbolic attack payload.

Recall that an attack payload is a string of bytes containing both concrete contents (e.g., fixed vulnerable return address) and variant contents (e.g., arbitrary shellcode, random padding). When a payload is being composed during the symbolic execution of the attack script, we use symbolic strings to represent variant contents and use extended APIs to perform symbolic string operations, while keeping concrete values and operations as normal. Thus the dumped payload packet is a big symbolic string composed of a sequence of substrings, where each substring is either a concrete byte string or a symbolic string by itself. Formally, $S_{sym} = (s_1 s_2 \dots s_i \dots s_n)$, where $s_i \in \{S_{con}\} \cup \{S_{sym}\}$. In addition, we also embed $< sym\_label, length >$ in S_{sym} to keep the semantics and the possible length of the string, where the length is either a concrete or symbolic integer. As an example, Listing 3 shows the contents of the payload when being composed in Lines 18-23 of the example script. The final dumped payload is the same as the one in Line 23.

Fixed-content pattern. This pattern has two types, either a simple byte string or a regular expression (regex). When parsing the payload, for each concrete substring, we extract it as a byte string pattern, such as the 12-byte string in the payload of Line 23. For each symbolic substring, if it can be matched by a regex, we extract the regex as a fixed pattern. If no regex is found, we move on to the next substring. In practice, we keep a mapping between regex-matchable symbolic labels and the regexes. Currently, we mainly focus on using regexes on payload paddings to achieve precise matching. For instance, we map the symbolic label `sym_rand_alpha` to a regex pattern `[a-zA-Z]`.

Fixed-length pattern. In some cases, although the contents may vary, their lengths stay the same. Such case typically happens when using padding to meet the size requirement. To achieve precise matching, SymExeLayer keeps track of the payload length during the composition. When parsing the payload, we identify the symbolic substrings with fixed lengths and extract them as patterns. When executing the example script in SymExeLayer, we keep updating the payload length. Later when parsing `<sym_rand_alpha, 2917>` in the dumped payload, we produce a length-quantified regex `[a-zA-Z]{2917}` as shown in Listing 2.

Fixed-offset pattern. Due to the format of some network protocols, some payloads can be located only after certain offsets of the packets. For instance, some FTP-based attack packets have regular FTP commands, followed with overlong paths as payloads to launch overflow attacks. In such cases, since the network protocol of the output API is logged, by applying the packet format of the protocol, we extract the offset of the payload, which is a pattern for precise matching of the payload location.

3.4.2 Pattern Refining and Consolidating

As MetaSymploit automatically generates signatures in a large scale, there are two requirements for the quality of the signatures. First, we should avoid generating signatures only having patterns of common benign bytes or patterns of trivial bytes/regexes, which may otherwise cause false positive. Second, we should avoid generating duplicated signatures with the same pattern set, which may cause useless redundancy and confuse the IDS.

First requirement. When a payload is finally sent through the output API, common benign bytes are introduced by network protocols as concrete substrings in the payload packet, including default protocol bytes (e.g., "Content-Type:text/html") and delimiter bytes (e.g., "\r\n"). To identify them, for each protocol, we keep a list of benign bytes. Based on the packet format, we examine the concrete substrings to search for the occurrences of benign bytes. If found, we strip the benign part and focus on the rest bytes for pattern extraction.

In addition, it is also important to avoid generating signatures only using trivial patterns such as too short byte string or too general regex patterns. Thus, we set a

threshold of minimum byte string length (e.g., >= 10) and a list of critical regexes (e.g., NOP regex [\x90]*). Given a set of extracted patterns, we generate signatures only if we can find at least one pattern whose length is above the threshold or whose regex is critical. Note that both the threshold and the critical regex list are adjustable. Security analysts can also define different thresholds and lists for different network protocols.

Second Requirement. Recall that SymExeLayer explores different execution paths in an attack script and dumps payloads in each path. Sometimes, two paths may differ only in a branch that is irrelevant to the payload content, thus finally composing the same payloads. Furthermore, two attack scripts may also share the same patterns. To consolidate the same patterns from different payloads into one signature, we keep a key-value hash map where each key is a pattern set and each value is a set of different payloads with the same pattern set. When a new payload is parsed, if its pattern set already exists in the hash map, we add this new payload, particularly its behavior & constraint log into the corresponding value set. The payloads and the logs in one set are analyzed together to generate only one signature.

3.4.3 Pattern Context Deriving

Apart from pattern extraction, it is equally important to provide the context of the patterns. The pattern context shows the insight into the attack script, such as what attack behavior and attack payload the patterns represent. It also gives security analysts the guidance on how to use the patterns, such as which target version and what OS environment the patterns can be used to protect.

Therefore, we analyze the behavior & constraint log to derive the pattern context. Since attack behaviors are captured as behavioral APIs in the log, we derive the context by translating the behavioral APIs into human-readable phrases. Some APIs have straightforward names, which can be simply translated into the description phrase (or even directly used), such as `probe_ver => Version Probing`. Others may not be intuitive. Particularly, certain behavior cannot be shown from a single API but a series of API calls. In such case, we group these API calls together as one behavioral pattern. When such pattern is found in the log, we translate it into the matched behavior name, such as `shellcode + get_target_ret => Stack Overflow`.

Sometimes, sophisticated attack scripts may have unprecedented behaviors whose APIs do not match any patterns. In such cases, we keep the derivable context while highlighting underived behavioral APIs in the log to help security analysts discover new attack behaviors. In fact, we use this technique in our prototype to collect patterns.

In regard to attack constraints, since the involved symbolic values represent attack conditions of each execution path, we retrieve the external source names in the symbolic labels and bind them with the conditions derived from the constraints (e.g., Target Version: 5).

Finally, when both the extracted pattern set and the derived context are ready, SigGen combines two together and generates a signature, which can be used to detect the payloads associated with this specific pattern set.

4 Implementation

We implement a prototype of MetaSymploit as a practical analysis tool targeting the Ruby-based attack framework Metasploit. Given a Metasploit attack script, our tool quickly analyzes it and automatically generates Snort rules as signatures that can defend against this specific script. Particularly, we developed a lightweight Ruby symbolic execution engine designed for attack script analysis. Powered by the engine, we build SymExeLayer on top of the launching platform of Metasploit. In this section, we first describe how the engine is designed and then explain how to adapt the engine for Metasploit.

4.1 A Lightweight Symbolic Execution Engine for Ruby

Traditionally, developing a symbolic execution engine requires heavy modification of the interpreter, which causes great engineering effort since Ruby has multiple active versions and interpreters (e.g., 1.8/1.9/2.0). However, we discover a new way to design a lightweight engine without modifying the interpreter. The engine is developed purely in Ruby (9.3K SLOC) as a loadable package compatible with multiple versions of Ruby. Thus it supports analyzing attack scripts written in different versions. Specifically, our engine has two modules: (1) a symbolic library that introduces rich symbolic support into Ruby; (2) a symbolic execution tracer that performs symbolic execution based on the actual script execution.

4.1.1 Library of Symbolic Support

The symbolic library realizes the functionality of *Symbolic API Extension*. The library introduces symbolic classes to hold symbolic values (e.g., `SymbolicString`, `SymbolicInteger`). To be transparent to attack scripts, we develop the same APIs in the symbolic classes as their concrete counterparts. On the other hand, we also extend indirect-variant-related APIs in the concrete classes to support handling symbolic arguments, so that concrete and symbolic objects can operate with each other.

Notably, `SymbolicString` class plays the key role in representing attack payloads. To hold the con-

tents, `SymbolicString` has an internal ordered array, where each item is either a concrete substring, or a symbolic substring with the `<sym_label, length>` embedded. When a `SymbolicString` API is called, it first checks whether the original concrete operation is still applicable to the concrete substrings. If so, the API uses the original logic in `String` to operate the concrete substrings. Otherwise, the API treats the contents as symbolic substrings, and processes the internal string array as symbolic expressions. When a symbolic-extended `String` API is called with symbolic arguments, it handles concrete and symbolic substrings in the same way as above and returns a `SymbolicString` object.

Later when SymExeLayer is integrated with Metasploit, we further include the symbolic-extended APIs of Metasploit into the symbolic library.

4.1.2 Symbolic Execution Tracer

The symbolic execution tracer transforms normal script execution into symbolic execution. It also realizes the functionality of *Behavior & Constraint logging*. To this end, we develop three techniques based on three advanced language features in Ruby (& Python[3]).

(1) Fine-grained execution tracing. This technique traces the symbolic execution line-by-line in an attack script. It keeps track of every method call. It also explores different paths when executing branches. We develop it by enhancing a language feature called *Debug tracing function* with Control Flow Graph (CFG).

Debug tracing function is a step-by-step execution tracing facility used for debugging such as Ruby's `set_trace_func` (Python's `sys.settrace`). It captures three major events, *line, call, return*. The *line* event shows the number of the current executing line. The *call/return* event shows the name of the method being called/returned. Every time an event happens, *Debug tracing function* suspends the execution and calls a registered callback function for further event analysis.

We develop our callback function using the CFG of the attack script. Since the CFG holds both the source code and the control flows, it offers rich semantics for analyzing the execution details when parsing every event. When a *line* event happens, we locate the current line's source code in the CFG. Then we retrieve all call sites in the current line, which will be matched with the following *call/return* events happening in this line. Particularly, this tracing mechanism can log behavioral API calls when they are found in the call sites.

Our callback function also handles branches to explore different paths. When the *line* event reaches a symbolic branch, we evaluate the branch source code and consult a constraint solver for both true and false branch constraints. If a solution exists, we concretize the symbolic branch condition to guide the interpreter to the desired branch (explained next). If both branches can be satisfied, we `fork` the script execution process into two processes to trace both branches. Otherwise, if no solution is returned, we terminate the execution process. Particularly, if attack constraints are found, the callback function would perform constraint logging.

(2) Runtime symbolic variable manipulation. This technique leverages the *Runtime context binding* language feature to manipulate the runtime values of symbolic variables. In particular, it inspects the values of attack payloads during composing. It also concretizes symbolic branch conditions to guide branch execution.

Runtime context binding can inspect and modify the runtime states of the script, such as Ruby's `Binding` and Python's `inspect`. It provides a *context* object that binds the runtime scope of the current traced code. The callback function can use this object to access all variables and methods in the scope of the traced code.

The first use of *context* is to inspect the runtime value of an attack payload when it is being composed. When a variable is detected to be assigned by payload composing APIs, the callback uses *context* to keep track of its value. The callback then logs the inspected values together with the payload composing APIs in the behavior log.

The second use of *context* is to guide symbolic branch execution. Since the interpreter cannot move forward with a symbolic condition, when the constraint solver returns a solution, for each symbolic variable in the condition, we use *context* to temporarily replace the symbolic value with the solved concrete value to guide the interpreter to the desired branch. Later when the *line* event shows that the branch is taken, we recover them back to their symbolic form. Recall the `version==5` in Listing 1. Since `version` is symbolic value, we temporarily replace its value with 5 to explore one branch, and uses a non-5 value for the other branch.

(3) Dynamic symbolic method wrapping. In some cases, the symbolic return values of method calls are not associated with any variables, thus cannot be manipulated using the second technique. To handle this, we leverage the *Dynamic method overriding* language feature to dynamically wrap the traced method, associate its return value with a temporary variable for manipulation.

Dynamic method overriding is a common feature in Ruby and Python that methods can be runtimely overridden and take effect immediately. Using this language feature, we dynamically create a wrapper method and override the original method right before the *call* event. Meanwhile, we also preserve the original method, and recover it right after the *return* event.

A more important use of the wrapping technique is to

[3]The techniques can also build an engine to analyze Python-based attack scripts, since Ruby and Python share many language features.

concretize symbolic methods in branch conditions. If no variable holds the symbolic return value of a method call in a branch condition, to guide symbolic branch execution, we override the symbolic method with the wrapper to return a solved concrete value. In practice some constraint solvers require the symbolic method calls to be associated with variables to enable the solving.

4.2 Adaptation for Metasploit

To analyze Metasploit attack scripts, we adapt the engine and the six techniques in both SymExeLayer and SigGen to work with the APIs provided by Metasploit and its built-in components.

The current prototype is based on Metasploit version 4.4 (released in Aug 2012). We select the top 10 most popular built-in components in Metasploit: `Tcp`, `Udp`, `Ftp`, `Http`, `Imap`, `Exe`, `Seh`, `Omelet`, `Egghunter`, `Brute`. The first 5 are popular network protocol components. The next 4 are used to attack Windows systems. `Exe` can generate exe file payloads. `Seh` can create SEH-based attacks. Both `Omelet` and `Egghunter` can compose staged payloads. The last `Brute` can create bruteforce attacks. These components cover 548 real attack scripts carried in Metasploit. By examining the APIs provided by the launching platform and these components of Metasploit, we perform three steps to adapt the engine for SymExeLayer and SigGen.

First, in the symbolic library, we apply symbolic API extension to the environment-related APIs such as `tcp.get`, `ftp.login`, `http.read_response`, and variant-payload-content-related APIs such as `rand_text`, `make_nops`, `gen_shellcode`[4]. The library also replaces the output APIs such as `ftp.send_cmd`, `http.send_request` with our local-dumping APIs. When the script calls these APIs during symbolic execution, SymExeLayer redirects the calls to the symbolic-extended APIs.

Second, to equip the symbolic execution tracer with behavior & constraint logging ability, we build a knowledge base collecting behavioral APIs such as `http.fingerprint`, `gen_egghunter` and keep a mapping between APIs and their behavior meaning for pattern context deriving. We also keep a list of symbolic labels for identifying attack constraints.

Third, based on the standards of the protocols and the implementation of the built-in components, we add the packet formats and common benign bytes of the five network protocols into the knowledge base. For instance, we develop specific parsers to parse payloads embedded in HTTP headers and FTP commands.

[4]The listed API names are abbreviated due to space limits. Note that Metasploit uses `payload` to represent shellcode. We use shellcode as a more general term to avoid confusion with attack payloads.

Note that both the API extension and the knowledge base are one-time system configuration. Since Metasploit components and their APIs are relatively stable for compatibility with various attack scripts, once they are collected and supported by MetaSymploit, newly distributed attack scripts that rely on these components can be directly supported and automatically analyzed.

5 Evaluation

We conduct our evaluation on an Intel Core i7 Quad 2.4GHz, 8GB memory, Ubuntu 12.10 machine. We run MetaSymploit based on Metasploit 4.4, using the official Ruby 1.9.3 interpreter. We evaluate our approach from three perspectives: (1) the percentage of real-world attack scripts that can be analyzed by MetaSymploit's symbolic execution; (2) the effectiveness of our automatically generated signatures to defend against real-world attacks; (3) the difference between our automatically generated rules and official Snort rules.

5.1 Coverage Testing with Symbolic Execution Engine

We first evaluate whether MetaSymploit can symbolically execute various attack scripts. We use MetaSymploit to analyze all 548 real attack scripts created with the top 10 popular Metasploit components. As the result shown in Table 1, 509 scripts (92.88%) are automatically executed by MetaSymploit in the symbolic mode without any manual modification of the scripts. Different attack conditions in the scripts are explored. The attack payloads are captured and Snort rules are generated.

In terms of analysis cost, since MetaSymploit reuses the launching platform of Metasploit on the official Ruby interpreter, the symbolic execution has almost the same speed as that Metasploit executes attack scripts normally (less than one minute on average). In fact, since the environment-related APIs are symbolic-extended, the time for real network communication is saved. Furthermore, signatures are generated in less than 10 seconds.

Among the remaining 39 scripts that MetaSymploit cannot automatically deal with, we encounter five main situations that deserve more discussion.

Loop with Symbolic Condition. We find that 9 scripts have conditional loops whose symbolic conditions cannot be solved by constraint solvers, which may cause infinite looping. As a common problem in classical symbolic execution, some previous approaches proposed using random concrete values to replace symbolic conditions to execute loops [20]. However, in our case, doing so may affect the precision of the payload contents. Other approaches such as LESE [35] specifically handle loops, which we plan to explore in future work.

Category	Num	Percentage	Require Manual Modification
Automatically Executed	509	92.88%	No
Symbolic Loop	9	1.64%	Avg 10 LOC/per script
Non-Symbolic-Extended API Call	12	2.19%	Avg 3 LOC/per script
Obfuscation & Encryption	13	2.37%	Not Supported
Multi-threading	3	0.55%	Not Supported
Bug in Scripts	2	0.37%	2 LOC in each script
Total Coverage	Auto 92.88%		All 96.90%

Table 1: The distribution of different situations in the symbolic execution of the 548 Metasploit attack scripts.

Currently, after manual analysis, we find that there are two cases of using the loops: byte-by-byte modifying a symbolic string whose length is a symbolic integer, and performing repeated attack steps in a bruteforce attack. In the first case, since the string length is not concrete, the looping rounds cannot be decided. However, we find no matter how many rounds are, the looping result is still a symbolic string. Therefore, we replace the loop code that operates the symbolic string with a new symbolic string to represent the looping result (10 LOC per script on average), while propagating the symbolic label and logging the loop information for further investigation.

In the second case, the `Brute` component provides an API that checks whether the target is compromised or not. It is typically used as a while loop condition. The loop keeps attacking the target until the API returns that the target is compromised. Since in our case the API returns a symbolic value as the target status, to avoid infinite looping, we set a counter with an upper bound in the extended version of this API, to control the looping rounds. If there are payloads and logs captured inside the loop, the differences between each round are analyzed to identify the constant patterns.

Non-Symbolic-Extended API Call. Due to the time limitations, other than the top 10 components, we have not symbolically extended other APIs in Metasploit. We detect 12 scripts that call the non-extended APIs related to assembly translating and encoding the payloads. Since very few APIs are involved, we decide to modify each of them individually at this time, and extend the entire components in future work. To handle these API calls, since `SymbolicString` supports payload content processing, when applicable to the concrete substrings, we allow the APIs to operate on the concrete parts, while preventing them from using the symbolic substrings, which may otherwise cause runtime errors. When the API operates on a pure symbolic string with no concrete substrings, we replace the API calls by creating new symbolic strings to represent the results of the API calls (3 LOC per script on average).

Obfuscation & Encryption. There are 13 cases with complicated obfuscation and encryption on the payload, where payload content processing is not feasible. Since the output of these operations is completely random, there is no constant pattern that can be extracted from the obfuscated or encrypted payload. Defending against obfuscation and encryption is an open question, which is beyond the scope of signature-based defense.

Multi-threading. Handling multi-threading is an advanced topic in symbolic execution. Existing research [37] explored the possibility by extending symbolic execution to handle multi-threaded programs. Currently, due to only 3 cases related to this situation, we plan to address this issue in future work.

Bug in Scripts. Interestingly, during the testing, we also discover 2 scripts with bugs that hang the execution when the script is generating a specific assembly code that jumps to the shellcode. From this result, we see that our approach is also useful for the purpose of finding bugs in attack scripts.

In summary, the percentage of scripts that are automatically handled is 92.88%. If the manually modified scripts are included, the percentage reaches 96.90%.

5.2 Effectiveness Validation using Real-world Attacks

To evaluate whether the automatically generated Snort rules can effectively detect real attacks, we use Metasploit attack scripts to attack 45 real-world vulnerable applications. These applications are acquired from `exploit-db.com`, a popular hacking website collecting attack scripts and free vulnerable applications. In all, there are 45 free vulnerable applications available in the website, with 45 corresponding Metasploit scripts. They include Java 7, Adobe Flash Player 10, Apache servers 2.0, Firefox 3.6, RealPlayer 11, multiple FTP servers such as Dream FTP, ProFTPD, VLC player 1.1, IRC servers and some less popular web-based programs.

We first use MetaSymploit to analyze the 45 attack scripts and automatically generate Snort rules. Then we set up two virtual machines, with one running Metasploit to simulate the attacker and the other running the vulnerable application as the vulnerable target. For each script, we choose two different shellcodes to launch two real attacks. To expose the entire attack flow, we allow the attack to compromise the target, and use Snort IDS 2.9.2 with our generated rules to detect attack payloads. Note

that due to the limited available versions of the applications, we focus on the rules of the attack payloads that target the application versions that we are able to obtain.

The initial results show that except the HTTP-based ones, all attack payload packets with both two types of shellcodes are correctly detected. Recall that our rules are based on the constant patterns of the payload, variant parts such as shellcodes do not affect the detection. But for Apache server attacks and Firefox attacks, our rules fail to catch the attack packets because the order of each HTTP header field is different from the one in our rules. Since the order of the HTTP header fields is not enforced by RFC definition, the extracted patterns from the HTTP header cannot be simply put into the signature in sequence. Therefore, we further improve our HTTP parser to handle each header field separately, to enable order-insensitive pattern matching. In the second round of testing, the HTTP-based attacks are also correctly detected.

Another interesting case is the Java 7 attack. In late Aug 2012, two days after a zero-day vulnerability in Java 7 was disclosed (CVE 2012-4681), a Metasploit attack script was distributed targeting this vulnerability [32]. At that time, we immediately used MetaSymploit to analyze this attack script and automatically generate a Snort rule based on the malicious jar payload composed by this script, and tested it in our environment. Our rule successfully detected the jar payload. Admittedly, there might be other ways different from the distributed Metasploit script to exploit the vulnerability. Nevertheless, our rule provides the first aid to the vulnerability without available security patch, to defend against attackers who directly use this widely-distributed script to launch attacks.

Apart from the effectiveness evaluation, we also use our rules generated from the 45 attack scripts to monitor normal network traffic, to investigate whether our rules would raise false positives on benign packets. We run the Snort with our rules in promiscuous mode to monitor the traffic of two Windows machines (Vista & 7) and a Ubuntu 12.04 machine. These machines are everyday-use machines in the CS department (no personal data is recorded). The monitoring is online for two months. No false positive is raised on benign packets. Such result is expected since our rules contain multiple specific patterns that matches only the Metasploit attack payloads. Appendix A shows a rule example for one of the 45 scripts.

5.3 Comparison with Official Snort Rules

To further assess the quality of the generated rules, we compare the MetaSymploit rules (MRs) of the 45 attack scripts with the recent Official Snort rules (ORs),

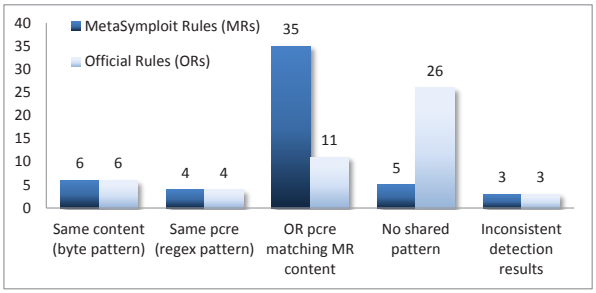

Figure 2: Pattern comparison between 53 MetaSymploit generated rules and 50 official Snort rules for 22 Metasploit attack scripts.

released in Nov 2012[5]. We use CVE number carried in both attack scripts and ORs to match each other. The result is surprising that only 22 attack scripts have corresponding ORs. The rest 23 are not even covered by ORs. This reveals a serious issue that existing defense is still quite insufficient compared to the fast spreading of public attack resources.

For the 22 officially covered scripts, there are 53 MRs and 50 ORs. In MetaSymploit, one script may have multiple rules detecting different payloads for different target versions. Whereas in the official rule set, one vulnerability may also have multiple rules detecting different ways that exploit it. By comparing the patterns in both rule sets, we summarize the result in Figure 2. We find that 44 MRs share patterns with 21 ORs. Specifically, 6 MRs and 6 ORs share the same `content` byte patterns. 4 MRs and 4 ORs share the same `pcre` regex patterns. Notably, 35 MRs have specific `content` that are matched with 11 ORs' general `pcre`. This is because the `pcre` regexes are generalized by security analysts based on large volumes of samples, while the `content` bytes (usually including vulnerable return addresses) are generated based on every attack payload of the scripts. An example is shown in Appendix A. Although in this case, the MR set is a subset of the OR one, we argue that as our goal is to defend against specific attack scripts, MRs give more insight of the attack payloads with more precise matching. Meanwhile, there are 5 MRs and 26 ORs with no pattern shared. This is because some vulnerabilities can be exploited in different ways, and the ORs have more patterns defined by analysts, while Metasploit scripts usually choose one way to exploit one vulnerability. Nevertheless, we still find that 2 scripts have 5 MRs whose patterns are not seen in ORs, which complement the OR set.

Besides, we also load the 50 ORs into Snort to test whether they can detect attacks launched by the 22 attack scripts. Interestingly, the result shows that only 17 scripts' attack payloads are detected, while no alert

[5] snortrules-snapshot-2922.tar.gz on www.snort.org/snort-rules/

is raised for the other 5 scripts. 2 scripts[6] are missed due to the lack of OR patterns as we mentioned above. The other 3 scripts, which have 3 MRs, are supposed to be detected by 3 corresponding ORs. After comparing these rules, we find the 3 ORs have some deficiencies that cause this inconsistent detection results. We list the detailed information of the 3 scripts and the deficiencies of the 3 ORs in Table 2. Note that some deficiencies are actually caused by inaccurate use of Snort rule flags such as the `http_uri`, `flow`. We find them by comparing these flags with the pattern context (e.g., Behaviors) in our rules. We have reported these discoveries to the official Snort team.

In sum, these results show that even the official Snort rules written by security analysts are incomplete and tend to be error-prone. MetaSymploit serves as a useful tool to complement and augment the existing IDS signatures by improving the completeness and the accuracy.

6 Discussion

Scenarios of using MetaSymploit signatures. As shown in the comparison (35 MRs vs 11 ORs), due to different pattern extracting mechanisms, ORs have less rules with more general patterns, while MRs have more rules with more specific patterns. It is possible that as the number of attack scripts is increasing, more and more signatures will be generated. If all signatures are loaded into the IDS, this may slow down the matching speed.

However, we argue that unlike ORs are used for general detection, MetaSymploit signatures should be used in two typical scenarios, which do not require loading all MRs in an IDS. First, as the goal of MetaSymploit is to provide quick defense against newly distributed attack scripts, the typical way of using our signatures is to give first aid to the vulnerable application without available patches to prevent attackers especially script kiddies using the new scripts to launch attacks (e.g., the Java 7 case). When the vulnerability is patched or the application is upgraded, our signatures can be removed from the IDS. Second, as the pattern contexts are embedded with the signatures, security analysts only need to deploy the signatures whose contexts are related to the protected environment or the protected target version, to avoid loading irrelevant signatures which may slow down the matching speed of the IDS.

Limitations. MetaSymploit inherits the limitations of classical symbolic execution. As we mentioned in Section 5.1, our current prototype requires manual analysis on handling complex symbolic loops. Recent approaches propose to use bounded iteration [21], search-guiding heuristics [40] and loop summary [22, 35] to address the loop issue. In MetaSymploit, different loop cases of attack scripts may require different techniques. For example, bounded iteration can be applied to handle loops of bruteforce attacks. Loop summaries can summarize the post-loop effect on symbolic payload contents. Search-guiding heuristics can help target payload-related loops to avoid getting stuck in irrelevant loops.

Apart from loops, path explosion is a more general issue related to performance and scalability. Too many paths in an attack script may prolong the analysis and delay the defense. In addition, it is possible that different paths in a script finally lead to the same attack payload output. Exploring these paths incurs extra efforts of pruning redundant payloads. Several techniques such as equivalent state tracking [9], state merging [26] and path partitioning [31] have been proposed to mitigate the path explosion issue. We plan to incorporate these techniques into MetaSymploit to avoid exploring paths that would compose redundant payload contents.

The limitations of constraint solvers may also affect the effectiveness of path exploration. Currently, we use Gecode/R [1] for solving integer/boolean constraints and HAMPI [23] for solving string constraints. In case when encountering complicated constraints (e.g., a non-linear constraint), the solvers cannot decide which branch to take. For the sake of completeness, we conservatively explore both branches, while marking the path constraints as uncertain in the log, which require more investigation by security analysts. Due to this fact, we regard our prototype as an assistant tool to reduce the workload of analysts, so that they only need to focus on complicated ones when facing large numbers of new attack scripts.

We envision possible attacks directly against MetaSymploit's defense mechanism. As MetaSymploit rules stick to the patterns in the distributed attack scripts, it is possible that experienced attackers may modify the distributed one to create new script variants without releasing them, which may evade the detection of MetaSymploit rules. Besides, experienced attackers may also try to exploit the limitation of symbolic execution when developing new scripts, such as introducing complex loops, non-linear constraints or even obfuscating the script code. However, both cases are non-trivial. They require advanced attack developing techniques, which are usually time-consuming and slow down the speed of developing and launching new attacks. In other words, with MetaSymploit, we raise the bar of the skill level and the time cost for developing and launching new attacks.

7 Related Work

Signature Generation. There has been a lot of work on automatic signature generation for malware defense. From the perspective of attacks, Autograph [24], Poly-

[6] `adobe_flash_sps.rb, mozilla_mchannel.rb`

Metasploit Script Name	CVE	Failure Reason of Official Snort Rules Missing Metasploit Payloads	Official Rule SID
badblue_ext_overflow.rb	2005-0595	The `http_uri` flag restricts the pattern searching in one header field, thus missing the Metasploit payload located in the following fields.	3816
sascam_get.rb	2008-6898	The `flow` pattern is set to check packets sent to the client while our pattern context shows the Metasploit payload is sent to the server.	16715
mozilla_reduceright.rb	2011-2371	The `content` byte pattern is wrong since it includes two variant bytes, which are randomly generated in the Metasploit payload.	19713

Table 2: The list of three Metasploit attack scripts which evade the detection from 3 Official Snort Rules

graph [29] and Hamsa [27] automatically generate worm signatures by extracting invariant contents from the network traffic of worms. Particularly, these approaches are based on the observation that even polymorphic worms have invariant contents that can be used as signature patterns. In MetaSymploit, we have the same observation when analyzing constant and variant payload contents composed by attack scripts. On the other hand, these approaches require collecting large amounts of malicious network traffic to identify invariant contents. However, this process is usually time-consuming and cannot provide quick defense against new attacks. In contrast, MetaSymploit does not need to collect any network traffic but only attack scripts, thus largely reducing the time of performing analysis and providing defense.

From the perspective of vulnerabilities, Vigilante [18], ShieldGen [19] and Bouncer [17] analyze vulnerable applications and their execution traces to generate signatures to block exploit inputs that can trigger the vulnerability. Brumley et al. [10, 11] also provide the formal definition of vulnerability-based signatures and propose constraint-solving-based techniques to generate such signatures. Elcano [13] and MACE [16] further use protocol-level concolic exploration to generate vulnerability-based signatures. Notably, program analysis techniques such as symbolic execution play an important role in these approaches as well as in MetaSymploit. But unlike these approaches, MetaSymploit only analyzes attack scripts without requiring the presence of vulnerable applications, thus avoiding the cost of obtaining various vulnerable applications or preparing various testing environments.

Symbolic Execution. Symbolic execution has been actively applied for security purposes [36]. BitBlaze [38] is a binary analysis platform based on symbolic execution. SAGE [21] uses dynamic symbolic execution to detect vulnerabilities in x86 binaries. EXE [14] and AEG [8] generate malicious inputs and exploits by symbolically executing vulnerable applications. Moser et al. [28] explore multiple execution paths for malware analysis. Since our analysis target, attack script is quite different from host-based binary level malware, the techniques proposed in these approaches such as memory inspection, system call analysis are not adaptable in our case.

Symbolic execution for scripting languages is still at early stage, due to the diversity of different kinds of scripting languages and various purposes of applications. Most work focuses on the web-based scripting languages, such as JavaScript [34], PHP [7, 41], and Ruby on Rails [15] web frameworks. Since these approaches are specifically designed for testing web applications (e.g., finding XSS and SQL Injection vulnerability), they are not applicable for analyzing general attack scripts and attack frameworks that target various vulnerable applications on different OS environments.

In particular, little work has been done for the symbolic execution of general-purpose scripting languages, such as Ruby and Python. PyStick [30] is an automated testing tool with input generation and invariant detection for Python. It is different from our purpose of using symbolic execution for security analysis. Bruni et al. [12] propose a library-based approach to develop symbolic execution. However, it uses only the dynamic dispatching feature, which limits symbolic execution only in primitive types. This limited functionality is insufficient for practical use.

8 Conclusion

Script-based attack frameworks have become an increasing threat to computer security. In this paper, we have presented MetaSymploit, the first system of automatic attack script analysis and IDS signature generation. MetaSymploit leverages security-enhanced symbolic execution to analyze attack scripts. We have implemented a prototype targeting the popular attack framework Metasploit. The results have shown the effectiveness of MetaSymploit in real-world attacks, and also the practical use in improving current IDS signatures.

9 Acknowledgements

We would like to thank the conference reviewers and shepherds for their feedback in finalizing this paper. This work is supported by the U.S. Army Research Office (ARO) under a MURI grant W911NF-09-1-0525, and also supported in part by an NSA Science of Security Lablet grant at North Carolina State Univer-

sity, NSF grants CCF-0845272, CCF-0915400, CNS-0958235, CNS-1160603.

References

[1] Constraint programming in ruby. http://gecoder.rubyforge.org/.

[2] The exploit database. http://www.exploit-db.com.

[3] Metasploit. http://www.metasploit.com.

[4] Arkeia backup client type 77 overflow (win32). http://www.metasploit.com/modules/exploit/windows/arkeia/type77, visited in Jan 2013.

[5] Conficker worm using metasploit payload to spread. http://blogs.mcafee.com/mcafee-labs/conficker-worm-using-metasploit-payload-to-spread, visited in Jan 2013.

[6] Top vulnerability exploit tools. http://sectools.org/tag/sploits, visited in Jan 2013.

[7] ARTZI, S., KIE, A., DOLBY, J., ERNST, M. D., KIEZUN, A., TIP, F., DIG, D., AND PARADKAR, A. Finding Bugs In Dynamic Web Applications. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA'08)*, pp. 261–272.

[8] AVGERINOS, T., CHA, S. K., LIM, B., HAO, T., AND BRUMLEY, D. AEG: Automatic Exploit Generation. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'11)*.

[9] BOONSTOPPEL, P., CADAR, C., AND ENGLER, D. RWset: Attacking Path Explosion in Constraint-Based Test Generation. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, pp. 351–366.

[10] BRUMLEY, D., NEWSOME, J., AND SONG, D. Towards Automatic Generation of Vulnerability-Based Signatures. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06)*, pp. 2–16.

[11] BRUMLEY, D., WANG, H., JHA, S., AND SONG, D. Creating Vulnerability Signatures Using Weakest Preconditions. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium (CSF'07)*, pp. 311–325.

[12] BRUNI, ALESSANDRO DISNEY, T. A Peer Architecture for Lightweight Symbolic Execution. Tech. rep., UC Santa Cruz, 2011.

[13] CABALLERO, J., LIANG, Z., POOSANKAM, P., AND SONG, D. Towards Generating High Coverage Vulnerability-Based Signatures with Protocol-Level Constraint-Guided Exploration. In *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection (RAID'09)*, pp. 161–181.

[14] CADAR, C., GANESH, V., AND PAWLOWSKI, P. EXE: Automatically Generating Inputs of Deathh. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS'06)*, pp. 322 – 335.

[15] CHAUDHURI, A., AND FOSTER, J. S. Symbolic Security Analysis of Ruby-on-Rails Web Applications. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*, pp. 585–594.

[16] CHO, C., BABIC, D., AND POOSANKAM, P. MACE: Model-inference-Assisted Concolic Exploration for Protocol and Vulnerability Discovery. In *Proceedings of the 20th USENIX Security Symposium* (2011).

[17] COSTA, M., CASTRO, M., AND ZHOU, L. Bouncer: Securing Software by Blocking Bad Input. In *Proceedings of twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP'07)*, pp. 117–130.

[18] COSTA, M., CROWCROFT, J., AND CASTRO, M. Vigilante: End-to-End Containment of Internet Worms. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*, pp. 133–147.

[19] CUI, W., PEINADO, M., WANG, H. J., AND LOCASTO, M. E. ShieldGen: Automatic Data Patch Generation for Unknown Vulnerabilities with Informed Probing. *Proceedings of the 2007 IEEE Symposium on Security and Privacy (S&P'07)*, 252–266.

[20] GODEFROID, P., AND KLARLUND, N. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*, pp. 213–223.

[21] GODEFROID, P., LEVIN, M. Y., AND BERKELEY, U. C. Automated Whitebox Fuzz Testing. In *Proceedings of Network and Distributed Systems Security (NDSS'08)*.

[22] GODEFROID, P., AND LUCHAUP, D. Automatic Partial Loop Summarization in Dynamic Test Generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA'11)*, pp. 23–33.

[23] KIEZUN, A., GANESH, V., GUO, P. J., HOOIMEIJER, P., AND ERNST, M. D. HAMPI: A Solver for String Constraints. In *Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA'09)*, pp. 105–116.

[24] KIM, H., AND KARP, B. Autograph: Toward Automated, Distributed Worm Signature Detection. In *Proceedings of the 13th USENIX Security Symposium* (2004), pp. 271–286.

[25] KING, J. C. Symbolic execution and program testing. *Commun. ACM 19*, 7 (1976), 385–394.

[26] KUZNETSOV, V., KINDER, J., BUCUR, S., AND CANDEA, G. Efficient State Merging in Symbolic Execution. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'12)*, pp. 193–204.

[27] LI, Z., SANGHI, M., CHAVEZ, B., CHEN, Y., AND KAO, M. Hamsa: Fast Signature Generation for Zero-day PolymorphicWorms with Provable Attack Resilience. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06)*, pp. 32–47.

[28] MOSER, A., KRUEGEL, C., AND KIRDA, E. Exploring Multiple Execution Paths for Malware Analysis. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy (S&P'07)*, vol. 0, pp. 231–245.

[29] NEWSOME, J., KARP, B., AND SONG, D. Polygraph: Automatically Generating Signatures for Polymorphic Worms. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy (S&P'05)*, pp. 226–241.

[30] NOTO-MONIZ, A. Software Agitation of a Dynamically Typed Language. Tech. rep., Worcester Polytechnic Institute, 2012.

[31] QI, D., NGUYEN, H. D., AND ROYCHOUDHURY, A. Path Exploration based on Symbolic Output. In *Proceedings of the 19th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE'11)*, pp. 278–288.

[32] RAGAN, S. Java zero-day added to blackhole exploit kit and metasploit. http://www.securityweek.com/java-zero-day-added-blackhole-exploit-kit-and-metasploit, visited in Aug 2012.

[33] RAMIREZ-SILVA, E., AND DACIER, M. Empirical Study of the Impact of Metasploit-Related Attacks in 4 Years of Attack Traces. In *Proceedings of the 12th Asian Computing Science Conference on Advances in Computer Science: Computer and Network Security (ASIAN'07)*, pp. 198–211.

[34] SAXENA, P., AKHAWE, D., HANNA, S., MAO, F., MCCAMANT, S., AND SONG, D. A Symbolic Execution Framework for JavaScript. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (S&P'10)*, pp. 513–528.

[35] SAXENA, P., POOSANKAM, P., MCCAMANT, S., AND SONG, D. Loop-Extended Symbolic Execution on Binary Programs. In *Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA'09)*, pp. 225–236.

[36] SCHWARTZ, E. J., AVGERINOS, T., AND BRUMLEY, D. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (S&P'10)*, pp. 317–331.

[37] SEN, K., AND AGHA, G. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools. In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV'06)*, pp. 419–423.

[38] SONG, D., BRUMLEY, D., YIN, H., AND CABALLERO. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Proceedings of the 4th International Conference on Information Systems Security (ICISS'08)*, pp. 1–25.

[39] SOPHOSLABS. Exploring the blackhole exploit kit. http://nakedsecurity.sophos.com/exploring-the-blackhole-exploit-kit, visited in Jan 2013.

[40] XIE, T., TILLMANN, N., DE HALLEUX, J., AND SCHULTE, W. Fitness-Guided Path Exploration in Dynamic Symbolic Execution. In *IEEE/IFIP International Conference on Dependable Systems & Networks (DSN'09)*, pp. 359–368.

[41] XIE, Y., AND AIKEN, A. Static Detection of Security Vulnerabilities in Scripting Languages. In *Proceedings of the 15th USENIX Security Symposium* (2006), pp. 179–192.

Appendix A Example of Rule Comparison

```
1  def exploit
2    ...
3    trigger = '/ldap://localhost/%3fA%3fA%3
         fCCCCCCCCCC%3fC%3f%90'
4    # Sending payload
5    send_request_raw({
6      'uri' => '/' + rewrite_path() + trigger +
7        shellcode(),
8      'version' => '1.0',
9      }, 2)
10   ...
11 end
```

Listing 4: The code snippet from a Metasploit attack script `apache_mod_rewrite_ldap.rb`

```
alert tcp any any -> any 80 (
  msg:"Metasploit apache_mod_rewrite_ldap,
  Target:[Apache 1.3/2.0/2.2],
  Behavior:[HTTP request with Vul-specific
      bytes]";
  content:"GET";
  content:"/ldap|3A|//localhost/%3fA%3fA%3
      fCCCCCCCCCC%3fC%3f%90";
  content:"|20|HTTP/1.0|0D 0A|Host|3A 20|";
  reference:cve,2006-3747;
  sid:5000539; rev:0;)
```

Listing 5: One MetaSymploit Rule (MR) for an attack payload of `apache_mod_rewrite_ldap.rb`.

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 80 (
  msg:"WEB-MISC Apache mod_rewrite buffer
      overflow attempt";
  content:"GET";
  content:"ldap|3A|";
  pcre:"/ldap\x3A\x2F\x2F[^\x0A]*(%3f|\x3F)[^\
      x0A]*(%3f|\x3F)[^\x0A]*(%3f|\x3F)[^\x0A
      ]*(%3f|\x3F)/smi";
  reference:cve,2006-3747;
  sid:11679; rev:5;)
```

Listing 6: One Official Snort Rule (OR) related to the Metasploit attack script in Listing 4.

In Appendix A, we give a simple example to illustrate the comparison between an official Snort rule containing general patterns with a MetaSymploit rule containing specific patterns.

Listing 4 shows the code snippet of the `exploit` method in the Metasploit attack script `apache_mod_rewrite_ldap.rb`. The script launches the attack by sending an HTTP GET request packet that contains a special URI byte string to trigger the vulnerability. Here `send_request_raw` is a Metasploit HTTP output API method that is symbolically extended by MetaSymploit to dump the entire payload packet.

Listing 5 is a MetaSymploit Rule (MR) based on the attack payload composed by the script. It contains the constant byte string patterns, especially the vulnerability triggering string that can identify the specific payload packet. Listing 6 is the corresponding Official Rule (OR) based on CVE matching. It contains a regular expression (regex) pattern generalized by security analysts based on large amounts of samples.

According to the Snort rule manual, a rule can have multiple `content` byte string patterns. By default, given a packet, Snort searches these `content` patterns in order. A rule can also have one `pcre` regex pattern. Snort searches the entire packet for the `pcre` pattern.

In the example rules, the first `content` in both rules share the same pattern "`GET`". The second `content` of the MR captures the triggering string, which includes the second `content` of the OR "`ldap|3A|`" as a substring. Furthermore, the second `content` of the MR is also matched by the general `pcre` regex of the OR. In addition, there is another `content` in the MR that captures the HTTP protocol version of the packet.

Although both rules can detect the attack payload of this script, the MR has multiple specific patterns that can precisely pinpoint the attacks launched by this script, thus having very low false-positive rate compared to the general OR. In practice, the MRs can help identify what attack scripts are used by attackers, providing a way for the defense side to profile and obtain more knowledge of the attacker side.