Context – Free – Language Parser for Advanced Network Intrusion Detection

Alvise Costa

March 29, 2010

Contents

Chapter 1

Introduction

Intrusion detection is the activity of detecting unauthorized intrusions performed by an *attacker* onto a *computer network/system*. Hence, an Intrusion Detection Systems (IDS) is a software and/or hardware designed to detect unwanted attempts at accessing, manipulating or disabling computer systems, mainly through a network such as the Internet.

To be more specific, an IDS is a specialized tool that can read and interpret the contents of log files from routers, firewalls, servers or other network devices and uses this informations to detect and deflect several types of malicious behaviors which could compromise the security of a computer system. Moreover, some types of IDSs can operate in *real–time* mode, that is they can detect an attack as it is performed without the necessity of logs [2].

An IDS is usually composed of two main components:

- *Sensors* that generate security events and pass them to the internal *detection engine*, which records events logged by the sensors in a database and uses a system of rules to generate alerts from security events received;

- a *Management Console* to monitor events and alerts and control the sensors;

There are many ways to categorize an IDS depending on the types and location of the sensors and the methodology used by the engine to generate alerts.

1.1 Types of Intrusion Detection Systems

It is possible to distinguish IDSs by the kinds of activities, traffic, transaction, or system they monitor. An IDS can be classified by its functionality into one of the following three main categories [2] [9]:

- Network – Based Intrusion Detection System (NIDS), an independent platform that identifies intrusion by examining network traffic and monitors multiple hosts.
- Host – Based Intrusion Detection System (HIDS), an agent on a host which identifies intrusions by analyzing system calls, applications logs, file – system modifications and other host activities and state.
- Distributed Intrusion Detection System (DIDS), a group of IDSs functioning as remote sensors and reporting to a central management station.

In practice, a combination of network, and host, and/or application–based IDS systems are used.

IDSs can also be categorized according to their differing approaches to event analysis. There exist two main models used for detecting attacks [3]:

- Misuse – or Signature – based IDSs that examine network traffic for pre–configured and predetermined attack patterns known as *attack signatures* or *exploit signatures*. This technique is called *misuse detection*
- Anomaly – based IDSs that use rules or predefined concept about "normal" and "abnormal" system activity to distinguish anomalies from normal system behavior and to monitor, report on, or block anomalies as they occur. This technique is called *anomaly detection*.

Most of the IDSs used today are signature – based. This is mainly due to the fact that they are easier to implement, configure and maintain than anomaly – based.

1.2 Signature – Based IDSs

A NIDS monitors an entire network segment and can also monitor all communications on that network segment. In a signature – based IDS, the detection

engine attempt to match the packet data against a set of pre–defined patterns called *signatures*. A signature is a pattern, usually defined as a string or as a regular expression (more often a combination of both), whose aim is to specify a single attack or either an entire class of attacks. Thus, once the network data packets have arrived from the network card, they will be decoded and then processed by the detection engine usually by a pattern – matching algorithm.

By analogy, a signature – based IDS does for a network what an antivirus software does for files that enter a system. That is, signature – based IDSs, as well as antivirus software, use a signature database of well – known attacks to match against to the current input data: a successful match will trigger an alert. Nevertheless, similarly to an anti – virus software, which fails to detect brand new viruses, a signature – based IDS fails to detect brand new attacks.

As cyber – attacks diversify, signature – based IDSs are likely to miss an increasingly part of attack attempts. However, most of the IDSs in use today are signature – based. The reason for this is that — also according to Kruegel and Toth [7] — a signature – based IDS is easier to implement and simpler to configure and maintain than an anomaly – based IDS, i.e. it is easier and less expensive to use.

1.2.1 Limitations of signature – based IDSs

As mentioned before, signature – based IDSs suffer of some limitations [3].

System upgrade and signatures development

Despite their ease of use, the task of upgrading the system is such important to the extent that it could seriously weights upon the proper functioning of the system itself. One of the motivation for the importance of this task is the fact that, almost every day, new security threats are discovered and new rules has to be developed (another motivation is that existing rules have to be patched).

Developing a signature is also a troubling task. Once a new attack is performed, and then made public, developers first need to carefully analyze it. The attack could exploit a well – known vulnerability or a new one. A signature should be developed in order to detect the way an attack exploits a given vulnerability rather than the specific attack payload. The reason for

this is that it is easy for an attacker to modify the attack payload without diminishing its effectiveness.

False Positives\False Negatives

Abstracting an attack is not always possible and it usually requires some efforts. It is very difficult to find a good agreement between a specific signature, which is not able to detect a simple attack variation, and a general signature, which will classify legitimate network traffic as an attack attempt. In other words, another limitation of signature – based IDSs is the *false positive\false negative rate*.

False positives occur when the IDS raises a warning when it should not. Basically, a false positive is a false alarm. The user might spent some time in properly tuning the system in order to decide what is relevant to his network and what is not.

On the opposite end, a false negative happens when the IDS do not detect an intrusion. This limitation is mainly due by the previously mentioned limitation of signature – based IDSs, which cannot detect unknown attacks until the ruleset has not been updated.

Leak of semantics in regular – expressions – based signatures

As we mentioned before, despite their ease of use and flexibility, regular – expressions – based signatures are likely to miss an increasingly part of attacks attempts. As we will show, this is mainly due to the fact that regular expressions are not so powerful to check for semantics in the attacks patterns.

In other words, regular expressions cannot detect the *context* of the pattern they are checking because they can only be used for pattern – matching. That is, if a pattern is matched, according to a certain regular – expression, we cannot state whether that match is only a coincidence or it is a real threat, because we have no information about the context we are working in.

1.2.2 Some Examples

Polymorphic viruses

One of the tasks that is not properly attended by attack signatures is the detection of *polymorphic* viruses. These kinds of viruses attack by executing the core instructions differently in subsequent generations.

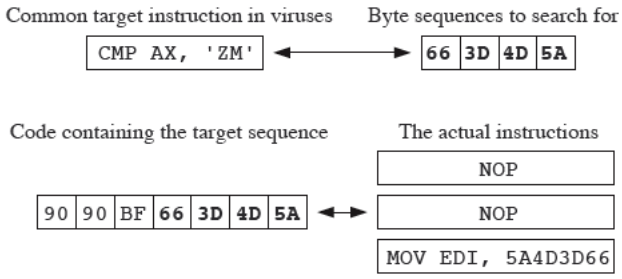

Figure 1.1: When disassembled, the code does not contain the target instruction. Thus, the false positives can occur when only pattern matching engine is used.

As the example illustrated in Figure 1.1 shows, simple pattern search can be ineffective or prone to false positives for such attack since sequence of bytes are different based on the locations and the content of the inserted codes. Referring to Figure 1.1, code segment A is a common target instruction in viruses. Segment B shows a possible byte sequence to look for when seeking the presence of the virus. Segment C shows the code containing the actual sequence been sought. The actual instructions D shows that segment as part of a NOP (null – operation) instruction. Missing, however, is the target instruction. This leads to false positive alerts when only pattern matching is used.

Injection Attacks

Injection attacks have become very common in the last few years together with the wide developing of dynamic web sites. This kinds of attacks are performed against the web application layer. The two attack techniques that are mainly and widely used are SQL Injection and Cross – Site Scripting. SQL Injection refers to the technique of inserting SQL meta – characters and commands into web – based input fields in order to manipulate the execution of the back – end SQL queries. Cross – Site Scripting attacks work by embedding script tags in URLs and "attracting" unsuspecting users to click on them, thus ensuring that the malicious script — usually Javascript — gets executed on the victim's machine.

Figure 1.2 shows an example of signature used by a modern IDS in order to detect SQL Injection attacks. This complex filter, based on regular expressions, will catch the widely used attack pattern " `' or 1=1-- -` " but

will miss another, though equally threatening, pattern: “ ’ or 2=2-- - ”. Again, simple pattern matching reveals to be ineffective and, in this case, prone to false negatives.

```
SecRule REQUEST_FILENAME|ARGS|ARGS_NAMES|REQUEST_HEADERS|XML:/*|!REQUEST_HEADERS:Referer "@pm sys.user_triggers sys.user_objects @@spid msysaces
instr sys.user_views sys.tab charindex sys.user_catalog constraint_type locate select msysobjects attnotnull sys.user_tables sys.user_tab_columns
sys.user_constraints waitfor mysql.user sys.all_tables msysrelationships msyscolumns msysqueries" \
        "phase:2,t:none,t:urlDecodeUni,t:htmlEntityDecode,t:lowercase,t:replaceComments,t:compressWhiteSpace,pass,nolog,skip:1"
SecAction phase:2,pass,nolog,skipAfter:959007
SecRule REQUEST_FILENAME|ARGS|ARGS_NAMES "(?:\b(?:(?:s(?:ys\.(?:user_(?:(?:t(?:ab(?:_column|le)|rigger)|object|view)s|c(?:onstraints|atalog))|
all_tables|tab)|elect\b.{0,40}\b(?:substring|ascii|user))|m(?:sys(?:(?:queri|ac)e|relationship|column|object)s|ysql\.user)|c(?:onstraint_type|
harindex)|waitfor\b\W*?\bdelay|attnotnull)\b|(?:locate|instr)\W+\()|\@\@spid\b)" \
        "phase:2,capture,t:none,t:htmlEntityDecode,t:lowercase,t:replaceComments,t:compressWhiteSpace,ctl:auditLogParts=+E,log,auditlog,msg:'Blind
SQL Injection Attack',id:'950007',tag:'WEB_ATTACK/SQL_INJECTION',logdata:'%{TX.0}',severity:'2'"
SecRule REQUEST_HEADERS|XML:/*|!REQUEST_HEADERS:Referer "(?:\b(?:(?:s(?:ys\.(?:user_(?:(?:t(?:ab(?:_column|le)|rigger)|object|view)s|c(?:onstraints|
atalog))|all_tables|tab)|elect\b.{0,40}\b(?:substring|ascii|user))|m(?:sys(?:(?:queri|ac)e|relationship|column|object)s|ysql\.user)|
c(?:onstraint_type|harindex)|waitfor\b\W*?\bdelay|attnotnull)\b|(?:locate|instr)\W+\()|\@\@spid\b)" \
        "phase:2,capture,t:none,t:urlDecodeUni,t:htmlEntityDecode,t:lowercase,t:replaceComments,t:compressWhiteSpace,ctl:auditLogParts=
+E,log,auditlog,msg:'Blind SQL Injection Attack',id:'959007',tag:'WEB_ATTACK/SQL_INJECTION',logdata:'%{TX.0}',severity:'2'"

#SecRule REQUEST_FILENAME|ARGS|ARGS_NAMES "\b(?:benchmark|encode)\b" \
#        "phase:2,chain,t:none,t:htmlEntityDecode,t:compressWhiteSpace,t:lowercase,ctl:auditLogParts=+E,log,auditlog,msg:'Blind SQL Injection
Attack',id:'950903',tag:'WEB_ATTACK/SQL_INJECTION',logdata:'%{TX.0}',severity:'2'"
#SecRule REQUEST_FILENAME|ARGS|ARGS_NAMES "t:none,t:htmlEntityDecode,t:compressWhiteSpace,t:lowercase,(?:[\\\(\)\%#]|--)"
#SecRule REQUEST_HEADERS|XML:/*|!REQUEST_HEADERS:Referer "t:none,t:urlDecodeUni,t:htmlEntityDecode,t:compressWhiteSpace,t:lowercase,\b(?:benchmark|
encode)\b" \
#        "phase:2,chain,t:none,t:urlDecodeUni,t:htmlEntityDecode,t:compressWhiteSpace,t:lowercase,ctl:auditLogParts=+E,log,auditlog,msg:'Blind SQL
Injection Attack',id:'959903',tag:'WEB_ATTACK/SQL_INJECTION',logdata:'%{TX.0}',severity:'2'"
#SecRule REQUEST_HEADERS|XML:/*|!REQUEST_HEADERS:Referer "(?:[\\\(\)\%#]|--)" t:none

SecRule REQUEST_FILENAME|ARGS|REQUEST_HEADERS|XML:/*|!REQUEST_HEADERS:Referer "@pm substr xtype textpos all_objects rownum sysfilegroups sysprocesses
user_group sysobjects user_tables systables pg_attribute user_users user_password column_id attrelid user_tab_columns table_name pg_class
user_constraints user_objects object_type dba_users sysconstraints mb_users column_name atttypid object_id substring syscat user_ind_columns sysibm
syscolumns sysdba object_name" \
        "phase:2,t:none,t:urlDecodeUni,t:htmlEntityDecode,t:replaceComments,t:compressWhiteSpace,t:lowercase,pass,nolog,skip:1"
SecAction phase:2,pass,nolog,skipAfter:959904
SecRule REQUEST_FILENAME|ARGS "\b(?:(?:s(?:ys(?:(?:(?:process|tabl)e|filegroup|object)s|c(?:o(?:nstraint|lumn)s|at)|dba|ibm)|ubstr(?:ing)?)|user_(?:
(?:(?:constrain|objec)t|tab(?:_column|le)|ind_column|user)s|password|group)|a(?:tt(?:rel|typ)id|ll_objects)|object_(?:(?:nam|typ)e|id)|
pg_(?:attribute|class)|column_(?:name|id)|(?:dba|mb)_users|xtype\W+\bchar|rownum)\b|t(?:able_name\b|extpos\W+\())" \
        "phase:2,capture,t:none,t:htmlEntityDecode,t:replaceComments,t:compressWhiteSpace,t:lowercase,ctl:auditLogParts=+E,log,auditlog,msg:'Blind
SQL Injection Attack',id:'950904',tag:'WEB_ATTACK/SQL_INJECTION',logdata:'%{TX.0}',severity:'2'"
SecRule REQUEST_HEADERS|XML:/*|!REQUEST_HEADERS:Referer "\b(?:(?:s(?:ys(?:(?:(?:process|tabl)e|filegroup|object)s|c(?:o(?:nstraint|lumn)s|at)|dba|
ibm)|ubstr(?:ing)?)|user_(?:(?:(?:constrain|objec)t|tab(?:_column|le)|ind_column|user)s|password|group)|a(?:tt(?:rel|typ)id|ll_objects)|object_(?:
(?:nam|typ)e|id)|pg_(?:attribute|class)|column_(?:name|id)|(?:dba|mb)_users|xtype\W+\bchar|rownum)\b|t(?:able_name\b|extpos\W+\())" \
        "phase:2,capture,t:none,t:urlDecodeUni,t:htmlEntityDecode,t:replaceComments,t:compressWhiteSpace,t:lowercase,ctl:auditLogParts=
+E,log,auditlog,msg:'Blind SQL Injection Attack',id:'959904',tag:'WEB_ATTACK/SQL_INJECTION',logdata:'%{TX.0}',severity:'2'"
```

Figure 1.2: An signature for SQL Injection Attacks

1.3 Current Alternatives to Signatures

Many solutions have been proposed by researchers and security teams, in order to overcome signatures limitations. So far, efforts to contrast these defections seem to focus on two main areas [3]: *anomaly – based IDSs* and *vulnerability signatures*.

1.3.1 Anomaly – Based Intrusion Detection Systems

As mentioned before, an anomaly – based IDS works according to a model of "normal" data/usage patterns and uses a similarity metric to compare the current input with the model itself. As soon as the network\system behaves in "abnormal" way an anomaly is detected and the IDS will react according to model rules. Thus, an anomaly – based IDS has the ability to detect previously unknown attacks as soon as they occur. However, setting up an anomaly – based IDS usually requires expert personnel. Many parameters need to be carefully configured and furthermore, each anomaly – based IDS is also different from the others, making it difficult to migrate to a different system that better fits user requirements.

1.3.2 Vulnerability Signatures

An alternative approach is to use signature – based IDS with other, and more powerful, kinds of signature. A *vulnerability signature* describes the *class* of data that trigger a vulnerability on the system. Vulnerability signatures are exploit – generic because they focus on how the end host interprets the data, rather than how the particular exploits works, and thus can detect variations of well – known attacks. This kind of signatures are very powerful because they reduce false negative, but they have some problems as well.

A vulnerability signature is useful only with respect to a single vulnerability, that is it can detect all (or most of) the variations of a certain vulnerability but will miss all the other types of threats.

Vulnerability signatures usually employ protocol parsing to retrieve the semantic content of the communication. This feature allows vulnerability signatures to be both more general and more precise than exploit signatures. However, this comes at a high performance cost and it requires very good knowledge on protocol level communication to be properly managed.

1.4 Research Question

So far we stated that signature – based IDSs are the most deployed in production environments, and the reason for this is their ease of use.

At deployment time, a signature – based IDS requires little work to set up. It can be deployed almost with an out – of – the – box configuration. Users con perform a selection of needed signatures, a task known as *tuning*, to avoid future false alerts: e.g., a signature for the IIS web server is useless when only Apache – based installations are in use. Tuning is performed once, and then updated from time to time. Thanks to tuning, a signature – based IDS generally generates a low rate of false alerts.

Furthermore, signature – based IDSs offer great flexibility for users. Users can write custom signatures, to detect specific events and patterns, or refine existing rules to improve detection. Users can manipulate the IDS engine as they wish.

However, although the mentioned pattern matching filters can be powerful tools for finding suspicious packets in the network, they are not capable of detecting other higher – level characteristics that are commonly found in network attacks. Signature syntax is quite easy, since it is mainly based on regular expression, however it is not as much as powerful as the modern network attacks would require.

To be more specific, regular expression are a *regular language*, or *type 3 language* according to Chomsky hierarchy. Parser for this kind of language can only check for syntactical correctness, but most of the attacks thrown today need to be checked for their "malicious behavior" to be detected.

All this means, it would be useful for a signature – based Intrusion Detection System, to be able to define a rule with another, and more powerful, kind of syntax: e.g., an expression might be parsed and checked in order to find semantic correctness. The main idea is then to allow a network detection engine to recognize language structures described by context – free (or *type 2*) grammars. This type of grammars is commonly used to define high – level computer programming language. This advanced detection feature also gives to IDSs the possibility to introduce a *threat grade*, that will turn out to be very useful and will add even more flexibility for users.

1.4.1 How to improve performances

We hypothesize that the ability to detect an advanced feature such as the language structure can lead to more accurate and advanced forms of detection engines. In other words, if the malicious code structure, embedded in the packet payload, can be examined, it may be possible to classify and detect an entire class of attack – variants with one grammar.

Therefore, for the motivations mentioned above, we focus our work on Network – Based Intrusion Detection Systems and we propose an advanced network intrusion detection engine that uses a parser to check for attack patterns in the network packet stream.

Chapter 2

Attacks

As me mentioned before, in the last few years, attacks against the Web application layer have required increased attention from security professionals. This is because no matter how strong firewall rulesets are or how diligent patching mechanisms may be, if Web application developers haven't followed secure coding practices, attackers will walk right into the system through port 80.

The two main attack techniques that have been used widely are SQL Injection and Cross Site Scripting attacks. SQL Injection refers to the technique of inserting SQL meta-characters and commands into Web – based input fields in order to manipulate the execution of the back-end SQL queries. These are attacks directed primarily against another organization's Web server. Cross Site Scripting attacks work by embedding script tags in URLs and enticing unsuspecting users to click on them, ensuring that the malicious Javascript gets executed on the victim's machine. These attacks leverage the trust between the user and the server and the fact that there is no input/output validation on the server to reject Javascript characters.

In this chapter we will present some of the most used web – attack techniques and we will show how they can be exploited to evade the signatures of Snort, one of the most widely used intrusion detection and prevention system[1].

[1]For a more precise description of Snort functionalities refer to paragraph 4.1.2 or to Appendix A

2.1 SQL Injection

A SQL Injection attack leverages vulnerabilities to inject a SQL query / command as an input, possibly via web pages. It is a type of web hacking that requires nothing but port 80 and it attacks on the web application (like ASP, JSP, PHP, CGI, etc) itself rather than on the web server or services running in the OS.

Many web pages take parameters from web user, and make SQL queries to the database. It is the case of a common user – login web page that requires name and password and makes a SQL query to the database to check if a user has valid name and password. With SQL Injection, it is possible for an attacker to send crafted user name and/or password field that will change the SQL query and thus grant him something else.

An attacker only needs a web browser to perform this kind of attack. He will, most likely, look for pages that allow him to submit data, i.e: login page, search page, feedback, etc[2].

2.1.1 A standard SQL Injection attack

Let's consider the following SQL query:

```
SELECT * FROM Users
WHERE Username= $username AND Password= $password
```

A similar query is generally used from the web application in order to authenticate a user. If the query returns a value it means that inside the database a user with that credentials exists, then the user is allowed to login to the system, otherwise the access is denied. The values of the input fields are generally obtained from the user through a web form. Suppose we insert the following Username and Password values:

```
$Username = 1' OR '1' = '1
$Password =  1' OR '1' = '1
```

The query will be:

```
SELECT * FROM Users
WHERE Username= '1' OR '1' = '1' AND Password= '1' OR '1' = '1'
```

[2]A SQL Injection attack that exploits the widely – used "email me my password" link can be found at: http://unixwiz.net/techtips/sql-injection.html

Let's suppose that the values of the parameters are sent to the server through the GET method, and that the domain of the vulnerable web – site is www.example.com; then the request that we will carry out is:

```
http://www.example.com/index.php?username=1'%20or%20'1'%20=%20'
       1&password=1'%20or%20'1'%20=%20'1
```

After a short analysis we notice that the query returns a value (or a set of values) because the condition is always true (OR 1=1). In this way the system has authenticated the user without knowing the username and password.

Sometimes, HTML pages use POST command to send parameters to another ASP page. Therefore, an attacker may not see the parameters in the URL. However, he can check the source code of the HTML, and look for "FORM" tag in the HTML code, probably finding something like this:

```
<FORM action=Search/search.asp method=post>
<input type=hidden name=A value=C>
</FORM>
```

Everything between the `<FORM>` and `</FORM>` have potential parameters that might be useful (exploit wise).

2.1.2 Regular – Expressions – based signatures for SQL Injection

A trivial regular expression to detect SQL injection attacks is to watch out for SQL specific meta–characters such as the single–quote (`'`) or the double–dash (`--`). In order to detect these characters, the following regular expression may be used:

```
/(\')|(\-\-)|(#)/ix
```

We first detect the single-quote or the presence of the double-dash. These are SQL characters for MS SQL Server and Oracle, which denote the beginning of a comment, and everything that follows is ignored. Additionally, if we're using MySQL, we need to check for presence of the `#`. Finally, pcre modifiers 'i' and 'x' are used in order to match without case sensitivity and to ignore whitespaces, respectively.

The above regular expression could be added into a signature–based IDS ruleset. For example, a new Snort rule would be composed as follows:

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"
SQL Injection - Paranoid"; flow:to_server,established;uricontent:
".php";pcre:"/(\')|(\-\-)|(#)/i"; classtype:Web-application-
attack; sid:9099; rev:5;)
```

This rule will clearly trigger many false positives alerts because the characters we are trying to detect will often appear in normal network traffic.

In the previous regular expression, we detect the double–dash because there may be situations where SQL injection is possible even without the single–quote. Let's consider, for instance, a SQL query which has the `where` clause containing only numeric values. Something like:

```
SELECT value1, value2, num_value3 FROM database
WHERE num_value3=some_user_supplied_number
```

In this case, the attacker may execute an additional SQL query, by supplying an input like:

```
3; insert values into some_other_table
```

The above signature could be additionally expanded to detect the occurrence of the semi–colon as well. However, the semi–colon has a tendency to occur as part of normal HTTP traffic. In order to reduce the false positives from this, and also from any normal occurrence of the single–quote and double–dash, the above signature could be modified to first detect the occurrence of the `=` sign. As we see in the previous section, user input will usually occur as a GET or a POST request, where the input fields will be reflected as:

```
username=user_supplied_value&password=user_supplied_value
```

Therefore, the SQL injection attempt would result in user input being preceded by a `=` sign. The resulting modified regular expression would be something like:

```
/(=)[^\n]*((\')|(\-\-)|(;))/i
```

This signature first looks out for the `=` sign. It then allows for zero or more non–newline characters, and then it checks for the single–quote, the double–dash or the semi–colon. A typical SQL injection attempt of course revolves around the use of the single quote to manipulate the original query so that it always results in a true value. Most of the examples that discuss this attack use the string `1'or'1'='1`, as we saw before. However, detection of this string can be easily evaded by supplying a value such as `1'or2>1--`.

2.2 Cross – Site Scripting

Cross – Site Scripting (XSS) attacks are a type of injection problem, in which malicious scripts are injected into the otherwise benign and trusted web sites. XSS attacks occur when an attacker uses a web application to send malicious code, usually in the form of a browser – side script, to a different end user.

A typical scenario is that in which a web application gathers malicious data from the attacker and creates an output page for the unsuspecting end user containing that malicious data, in a manner to make it appear as a valid content from the web site. The data is usually gathered in the form of a segment of JavaScript, but may also include HTML, Flash or any other type of code that a browser may execute. The end user's browser has no way to know that the script should not be trusted, and will execute it. Because the browser "thinks" that the script came from a trusted source, the malicious script can access any cookies, session tokens, or other sensitive information retained by the browser and used with that web site. The variety of attacks based on XSS is almost limitless, but they commonly include transmitting private data like cookies or other session information to the attacker, redirecting the victim to web content controlled by the attacker, or performing other malicious operations on the user's machine under the guise of the vulnerable site.

Cross – site – scripting attacks can generally be categorized into two categories: stored and reflected.

2.2.1 Stored XSS Attacks

Stored attacks are those where the injected code is permanently stored on the target servers, such as in a database, in a message forum, visitor log, comment field, etc. The victim then retrieves the malicious script from the server when it requests the stored information.

Example

Let's consider a web – site that permits us to leave a message to other users. We inject a script instead of a message in the way described in Figure 2.1.

The server will store the "message" we sent. Now, when a user clicks on our fake message, his (or her) browser will execute our script as shown in Figure 2.2.

Figure 2.1: Injection of a "malicious" script

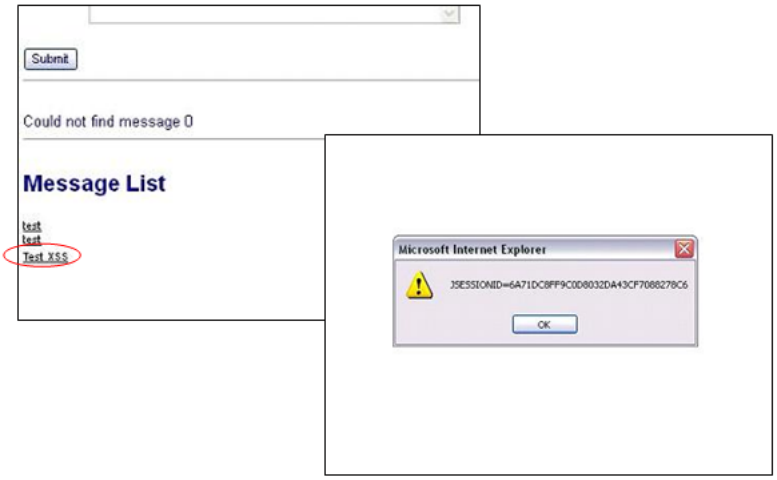

Figure 2.2: A user's click on the link will execute the "malicious" script

2.2.2 Reflected XSS Attacks

Reflected attacks are those where the injected code is reflected off the web server, such as in an error message, search result, or any other response that includes some or all of the input sent to the server as part of the request. Reflected attacks are delivered to victims via another route, such as in an e-mail message, or on some other web server. When a user is tricked into clicking on a malicious link or submitting a specially crafted form, the injected code travels to the vulnerable web server, which reflects the attack back to the users browser. The browser then executes the code because it came from a "trusted" server.

Example

The following JSP code segment reads an employee ID, `eid`, from an HTTP request and displays it to the user.

```
<% String eid = request.getParameter("eid"); %>
...
Employee ID: <%= eid %>
```

The code in this example operates correctly if `eid` contains only standard alphanumeric text. If `eid` has a value that includes meta–characters or source code, then the code will be executed by the web browser as it displays the HTTP response.

Initially this might not appear to be much of a vulnerability. After all, why would someone enter a URL that causes malicious code to run on their own computer? The real danger is that an attacker will create the malicious URL, then use e-mail or social engineering tricks to lure victims into visiting a link to the URL. When victims click the link, they unwittingly reflect the malicious content through the vulnerable web application back to their own computers.

2.2.3 Regular – Expression – based signatures for Cross Site Scripting

When launching a cross-site scripting attack, or testing a Website's vulnerability to it, the attacker may first issue a simple HTML formatting tag such as `<b>` for bold, `<i>` for italic or `<u>` for underline. Alternatively, he may try

a trivial script tag such as `<script>alert("OK")</script>`. This is likely because most of the printed and online literature on XSS use this script as an example for determining if a site is vulnerable to XSS. These attempts can be trivially detected.

The following regular expression checks for attacks that may contain HTML opening tags and closing tags `<>` with any text inside. It will catch attempts to use `<b>` or `<u>` or `<script>`. The regular expression is case – insensitive:

```
/(<)(\/)*[a-z0-9]+(>)/ix
```

We first check for opening angle bracket, then we look for the forward slash for a closing tag, then again we check for alphanumeric string inside the tag and finally we check for closing angle bracket.

The above regular expression could be added into a signature–based IDS ruleset. For example, a new Snort rule would be composed as follows:

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"
Cross-site scripting attempt"; flow:to_server,established;pcre:
"/(<)(\/)*[a-z0-9]+(>)/ix"; classtype:Web-application-attack;
sid:9000; rev:5;)
```

Cross-site scripting can also be accomplished by using the `<img src=>` technique. The following regular expression could be used in a Snort rule to detect attacks using this technique:

```
/(<)(i)(m)(g)[^\n]+(>)/ix
```

This signature first looks out for the opening angle bracket. Then it checks for the letters '`img`' followed by any characters other than a new line and finally it looks for a closing angle bracket.

The `<img src=>` technique is clearly not the only one an attacker can exploit. An attacker could use any HTML tag that permits him to insert malicious code inside them. For example, tags like `<frame>` or `<object>` could be used.

Thus someone could be forced to use a signature like the following one to protect his system against XSS attempts:

```
/(<)[^\n]+(>)/ix
```

The above "paranoid" signature simply looks for the opening HTML tag, followed by one or more characters other than the newline, and then followed by the closing tag.

Adopting such kind of signatures will grant the detection of anything that even remotely resembles a cross – site scripting attack, but will also end up with pretty much false positive alerts.

Unfortunately, most of the modern IDS, such as Snort, in order to avoid or reduce false positive alerts, supply default signatures for XSS that can be easily evaded.

Chapter 3

High – Level Language Recognition

A compiler is a program that can read a program in one language — the source language — and translate it into an equivalent program in another language — the target language. That is, computer program compilers map a source program into a semantically equivalent target program. This process can be, at first, divided into two parts: analysis and synthesis.

The analysis part breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program. If the analysis part detects that the source program is either syntactically ill formed or semantically wrong, then it must provide informative messages, so the user can take corrective action. The analysis part also collects information about the source program and stores it in a data structure called the "symbol table", which is passed along with the intermediate representation to the synthesis part.

The synthesis part constructs the desired target program from the intermediate representation and the information stored in the symbol table.

The analysis part is mainly focused on verifying and constructing software structure using the grammatical rules while the synthesis part is responsible for detecting semantic errors and checking type usage.

3.1 Context – Free Grammars

A context – free grammar (CFG) is a set of recursive rewriting rules (or *productions*) used to specify the syntax of a language. A context – free grammar has four components[1]:

1. A set of terminal symbols, sometimes referred to as "tokens". The terminals are the elementary symbols of the language defined by the grammar.

2. A set of nonterminals, sometimes called "syntactic variables". Each nonterminal represents a set of strings of terminals, in a manner we shall describe.

3. A set of productions, where each production consists of a nonterminal, called the head or left side of the production, an arrow, and a sequence of terminals and/or nonterminals, called the body or right side of the production. The intuitive intent of a production is to specify one of the written forms of a construct; if the head nonterminal represents a construct, then the body represents a written form of the construct.

4. A designation of one of the nonterminals as the start symbol.

The most common formal system for presenting such rules for humans to read is Backus – Naur Form or "BNF", which was developed in order to specify the language Algol 60. Any grammar expressed in BNF is a context – free grammar. This formal representation is more powerful than the regular expression. In addition to regular expression, it is able to represent more advanced programming language structures such as balanced parenthesis and recursive statements like if – then – else. Given such powerful formal representation, it may be possible to devise more efficient and accurate signature for advanced forms of attack.

A CFG for Arithmetic Expressions

An example grammar that generates strings representing arithmetic expressions with the four operators +, -, *, /, and numbers as operands is:

1. $< expression > \longrightarrow number$

2. $< expression > \longrightarrow (< expression >)$

3. $< expression > \longrightarrow < expression > + < expression >$

4. $< expression > \longrightarrow < expression > - < expression >$

5. $< expression > \longrightarrow < expression > * < expression >$

6. $< expression > \longrightarrow < expression > / < expression >$

The only nonterminal symbol in this grammar is $< expression >$, which is also the start symbol. The terminal symbols are {+, -, *, /, (,), number}. (We will interpret "number" to represent any valid number.)

The first rule (or production) states that an $< expression >$ can be rewritten as (or replaced by) a number. In other words, a number is a valid expression.

The second rule says that an $< expression >$ enclosed in parentheses is also an $< expression >$. Note that this rule defines an expression in terms of expressions, an example of the use of recursion in the definition of context – free grammars.

The remaining rules say that the sum, difference, product, or division of two $< expression >$s is also an expression.

3.1.1 Context – Free Grammars VS Regular Expressions

To understand the differences between regular expressions and context – free grammars, we can contrast the expression grammar described above against the following regular expression:

$$((ident|num)(+|-|*|/))^*(ident|num)$$

Both the regular expression and the CFG describe the same set of expressions. Regular expressions can be described by *regular – grammars* or, equivalently, by Finite Automata (FA). In a regular – grammar productions are restricted to two forms, either $A \longrightarrow a$, or $A \longrightarrow aB$, where A, B are nonterminal symbols and a is a terminal symbol.

In contrast, a context – free grammar allows productions with right – hand sides that contain an arbitrary set of terminal and nonterminal symbols. Thus regular regular grammars are a proper subset of context – free grammars.

It is trivial to notice that context – free grammars are strictly more powerful than regular expressions. More precisely:

- Any language that can be generated using regular expressions can be generated by a context-free grammar.
- There are languages that can be generated by a context-free grammar that cannot be generated by any regular expression.

As a corollary, CFGs are strictly more powerful than Deterministic Finite Automata (DFA) and Non – Deterministic Finite Automata (NDFA), which are the two types of Finite Automata used to describe Regular Languages.

The proof is in two parts:

- Given a regular expression R , we can generate a CFG G such that L(R) == L(G).
- We can define a grammar G for which there there is no FA F such that L(F) == L(G).

Where L(X) is any language generated by grammar X.

With the example of arithmetic expressions we showed that CFGs are at least as powerful as regular expressions. With the following example we will demonstrate that Context – Free Grammars are much more powerful and expressive with respect to Regular Expressions, focusing on one particular aspect of CFGs that is the fact that they can count.

A CFG with no corresponding RE

Finite Automata, and thus Regular Expressions, cannot count. That is no FA can recognize the language $\{0^n1^n | n >= 1\}$ (i.e., the set of strings containing one or more zeros followed by an equal number of ones).

Assume such an FA exists, and it has N states. What happens when the input string has N+1 zeros in it, followed by N+1 ones?

- Since the FA only has N states, we must visit some state sT twice on seeing N+1 zeros.
- The FA cannot know whether we are entering sT for the first time, when we've seen i ¡ N zeros, or the second time, when we've seen j ¿ i zeros.

- There must be a path from sT to an accepting state, since the input string is in the language.
- The FA will accept an input string without an equal number of zeros and ones, since i != j, and there is a path to an accepting state from sT on the remaining input.

This language is generated by the following CFG:

1. $< S > \longrightarrow 0 < S > 1$
2. $< S > \longrightarrow 01$

We can prove that this grammar generates the language by induction on n, the number of zeros and ones in the string.

1. For the basis step, n = 1, and the string is 01. This string is generated by applying the second production once.
2. For the inductive step, assume we can generate 0^n1^n. The last production applied must have been production 2, so the string must have been $0^(n-1) < S > 1^(n-1)$. If we apply production 1 and then production 2, we get $0^n < S > 1^n$, and then $0^(n+1)1^(n+1)$. Thus, we can generate all strings of the form $\{0^n1^n | n >= 1\}$.
3. Since we can only apply production 1 some number of times followed by production 2, these are the only strings generated by the grammar.

3.2 Language Processing

The compilation process operates as a sequence of phases, each of which transforms one representation of the source program to another. A typical decomposition of a compiler into phases is shown in Figure 3.1. The symbol table, which stores information about the entire source program, is used by all phases of the compiler.

The phase that is used for detecting tokens from regular expressions is called the lexical analysis or scanning. The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes. For each lexeme, the lexical analyzer

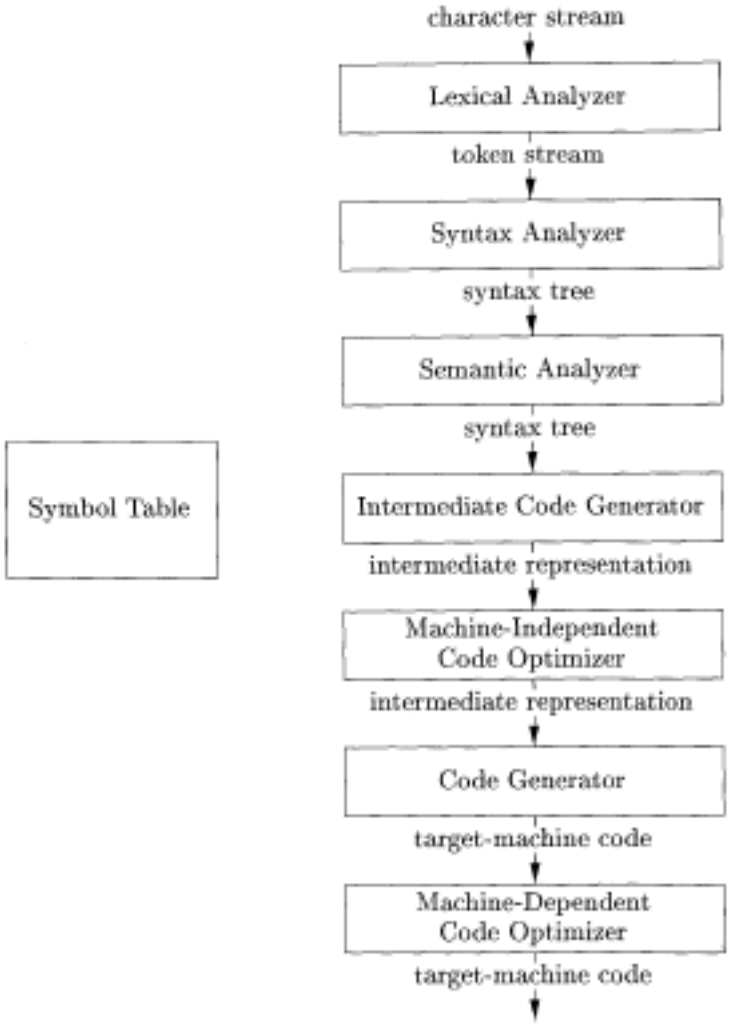

Figure 3.1: Phases of a Compiler

produces as output a token that it passes on to the subsequent phase, syntax analysis. Scanners can be automatically generated by tools such as `Lex` or `flex`.

The second phase of the compiler is syntax analysis or parsing. The parser uses the first components of the tokens produced by the lexical analyzer to create a tree – like intermediate representation that represents the grammatical structure of the token stream. A typical representation is a syntax tree in which each interior node represents an operation and the children of the node represent the arguments of the operation. For modern compilers, the parsers are automatically generated, according to the rules defined in the grammars, through tools such as `Yacc` or `Bison`.

The subsequent phases of the compiler use the grammatical structure to help analyze the source program and generate the target program. For our application we are only interested in syntactic acceptance, therefore we focus our research efforts on understanding lexical and syntactical analysis of the compilers.

3.2.1 Scanner

The first phase of language recognition is the conversion of sequence of bytes to sequence of predefined tokens. There are several similarities between a token scanner and the signature matcher of an IDS. Both systems are responsible for detecting and identifying predefined byte patterns from the stream of data input. The scanner is provided with a point in the input stream at which it is to produce sequence of tokens. Therefore, the token sequence produced by a lexical scanner is unique. On the other hand, a signature matcher does not constrain where the embedded string starts; it simply detects matching patterns as it scans the stream at every byte offset.

3.2.2 Parser

The parser obtains a string of tokens from the lexical analyzer and verifies that the string of token names can be generated by the grammar for the source language. Conceptually, for well – formed programs, the parser constructs a parse tree and passes it to the rest of the compiler for further processing.

There are three general types of parsers for grammars: universal, top – down and bottom – up. The methods commonly used in compilers can be

classified as being either top – down or bottom – up. As implied by their names, top – down methods build parse trees from the top (root) to the bottom (leaves), while bottom – up methods start from the leaves and work their way up to the root. In either case, the input to the parser is scanned from left to right, one symbol at a time.

The most efficient top – down and bottom – up methods work only for subclasses of grammars, but several of these classes, particularly, LL (suitable for top – down parsing) and LR grammars (suitable for bottom – up parsing), are expressive enough to describe most of the syntactic constructs in modern programming languages.

Chapter 4

System Architecture

Computer program source codes are analyzed with scanner and parser to determine correctness in the language structure. We propose to adapt the concept into the packet inspection system to effectively recognize structure within the network traffic.

Figure 4.1 is a block diagram of our advanced inspection process. After the header and the payload inspection, the pattern indices are converted to the streams of tokens by the scanner. The streams of tokens are then forwarded to the hardware parser to verify its grammatical structure. When the parser finds that the token stream conforms to the grammar, the packet can be marked to be suspicious.

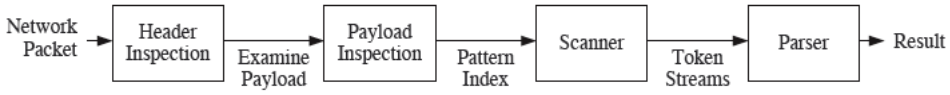

Figure 4.1: Processing phases of a network packet

Our aim is that of developing a high – performance software version of the system described above and to test it against well known classes of attack such as SQL Injection and Cross Site Scripting, for which there not exist, at present, any kind of properly working signature.

4.1 Network packets capture

As shown in Figure 4.1, the first issue we have to face in the development of our system is that of network packets capture. Packet capture is the act of capturing data packets crossing a network. Deep packet capture (DPC) is the act of capturing complete network packets (header and payload) crossing a network. Once captured and stored, either in short-term memory or long-term storage, software tools can perform Deep packet inspection (DPI) to review network packet data and, in our case, identify security threats.

Packet capture has the ability to capture packet data from the data link layer on up (layers 2 – 7) of the OSI model. This includes headers and payload. As it's been said already, headers include information about what is contained in the packet, while the payload includes the actual content of the packet and therefore is the part of the packet we are interested in for inspection. Complete capture encompasses every packet that crosses a network segment, regardless of source, protocol or other distinguishing bits of data in the packet. Thus, complete capture is the unrestricted, unfiltered, raw capture of all network packets. The capture of packets for which public visualization is not allowed, is called *sniffing*.

DPC devices usually have the ability to limit capture of packets by protocol, IP address, MAC address, etc. With the application of filters, only complete packets that meet the criteria of the filter (header and payload) are captured, diverted, or stored. For our purpose, as we are dealing with attacks directed to web applications, we limit our inspection on HTTP traffic, thus looking up for TCP traffic on port 80 of our dedicated web server.

At data link layer level there is a distinction between networks using packet switching and networks not using it. In non – switching networks, Ethernet packets go through every device on the network assuming that each device will accept only packets destined to itself. Nevertheless, is quite simple to set a network device in *promiscuous mode* so that it can examine all the packets independently from their destination address. To this extent, in Unix – like systems, we can use the `ifconfig` command, i.e. `ifconfig eth0 promisc`. Thus, promiscuous – mode packet sniffing on non – switching network, allows us to obtain any kind of useful information.

4.1.1 `libpcap` sniffer

A standard programming library called `libpcap` can be used in order to capture packets from a network segment, in a reliable and portable way. Program 1 is an example of how to use `libpcap` library to set up a very simple C – language – based sniffer.

Program 1 raw_sniffer.c

```
#include <pcap.h>
...

int main() {
    struct pcap_pkthdr header;
    const u_char *packet;
    char errbuf[PCAP_ERRBUF_SIZE];
    char *device;
    pcap_t *pcap_handle;
    int i;

    device = pcap_lookupdev(errbuf);
    if(device == NULL){
        ... //some logic for error managing
    }
    pcap_handle = pcap_open_live(device, 4096, 1, 0, errbuf);
    if(pcap_handle == NULL){
        ... //some logic for error managing
    }
    for(i=0; i<3; i++) {
        packet = pcap_next(pcap_handle, &header);
        ... //some logic for packet inspection
    }
    pcap_close(pcap_handle);
}
```

First thing to do is to include `pcap.h` header – file, that supplies several structures and definitions used by `pcap` functions. `pcap` functions use an error buffer in order to output error messages, thus `errbuf` variable represents

this buffer. Variable `header` is a `pcap_pkthdr` structure that contains extra informations about the packet (i.e. the capturing – date and its length). Pointer `pcap_handle` acts in a very similar way of a file – descriptor but it is used to refer to a packet – capture object. Function `pcap_lookupdev()` looks up for a proper device to sniff on, and returns a string – pointer, referring to a static – function memory area. The return value represents the first available device, if no proper interface is found, the return value is `NULL`. Function `pcap_open_live()` opens a device for packet capture and returns a handle to it. Arguments of this function are the sniffing device, packet maximum dimension, a flag for promiscuous mode, a timeout value and a pointer to the error buffer. Finally, the packet capture loop uses `pcap_next()` to capture next packet. This function returns a pointer to the actual packet. Then function `pcap_close()` closes the capture interface.

With a bit more effort we can write a sniffer that can decode packet – headers on different levels: Ethernet, IP, TCP, and so on. Furthermore, `libpcap` contains a function called `pcap_loop()` which supplies a better solution for packets capture, with respect to a simple loop on a `pcap_next()` function call. `pcap_loop()` utilizes a callback function; this means that it gets, as argument, a pointer to a function which will be called any time a packet has been captured.

Another issue we have to face developing our system is packet fragmentation and defragmentation, that is the action of disassembly and reassembly a packet (or a stream of packets). Packet fragmentation and defragmentation is one of the main jobs of the IP protocol. The IP protocol defines the maximum size of a packet as 64 KB, which comes from the fact that the "`len`" field of the header, which represents the size of the packet in bytes, is a 16-bit value. However, not many interface types can send packets of a size up to 64 KB. This means that when the IP layer needs to transmit a packet whose size is bigger than the Maximum Transmit Unit (MTU) of the egress interface, it needs to split the packet into smaller pieces. Regardless of how the MTU is computed, the fragmentation process creates a series of equal-size fragments. The Identification field, and Fragment offset field along with Don't Fragment and More Fragment flags in the IP protocol header are used for fragmentation and reassembly of IP datagrams. A fragmented IP packet is normally defragmented by the destination host, but intermediate devices or applications that need to look at the entire IP packet may have to defragment it, too. Unfortunately, `libpcap` has some limitations and does not supply any function that can help us with that.

4.1.2 Snort 3 Detection Engine

In order to exceed `libpcap` limitations, we also developed an *engine module* for Snort 3.0, the latest Snort implementation.

Snort 3.0 architecture essentially consists in two main components. The first component is the "Snort Security Platform 3.0" or SnortSP, which provides functionality such as loading, logging and event generation, data acquisition, decoding and validation, flow management and so on. The second major component are "engines" which basically are the analysis modules that plug into SnortSP. The basic idea in SnortSP is to instantiate data source, engine, analyzer and output modules, link them together and then start the engine. Once the engine is running we can manage it via the command shell interface[1].

In Snort 3 the data source is the set of subsystems that handle data acquisition (e.g. pcap, af_packet), decode, flow management and IP defragmentation. Data sources are a core element of SnortSP 3.0, they acquire traffic from the network, decode it, put it into the flow table subsystem and then hand it on to the rest of the analysis and reporting elements within a processing thread. For intrusion detection purposes, we need to link a data source to an engine in order to get some packet inspection capabilities.

Engines are the central organizing mechanism in SnortSP, an engine has a data source, analyzer(s), output manager and so on. An engine contains the control logic for interfacing data sources with analysis engines and managing data flow and the engine threads that perform any packet analysis.

Analyzers are the modules that analyze the packet data and generate the appropriate events. Outputs are modules that can be configured to output events generated by the analytic instances.

Unfortunately, Snort 3.0 is delivered with no documentation on how to properly program an engine module, but the C – code of a sample dummy engine is available. Thus, we "hacked" a bit that code in order to make our system work as an analytic module for SnortSP.

4.2 Deep packet inspection

As we can see in Figure 4.1, after packet capture, our system architecture begins the packet inspection phase, in order to detect any attack attempt.

[1]For a more precise presentation of Snort refer to Appendix A

In this phase, our application tries to accomplish its main task exploiting the idea we presented in Chapter 1, that is to utilize context – free – language parsing technologies for network intrusion detection.

As we said in Chapter 3[2], for our application we are only interested in lexical and syntactical analysis of the packet payload because we want to check for syntactic correctness. Thus, the "signatures" our application utilizes are split into two parts: the first part is used to perform lexical analysis, and the other one to perform syntactical analysis.

For the development of this part of the system, we take advantage of widely used automatic scanner generator `flex` and automatic parser generator `Bison`.

4.2.1 Automatic scanner generator `flex` [8]

`flex` is a tool for generating high – performance scanners. The `flex` program reads the given input files for a description of a scanner to generate. The description is in the form of pairs of regular expressions and C code, called rules. `flex` generates as output a C source file. This file can be compiled and linked with the `flex` runtime library to produce an executable. When the executable is run, it analyzes its input for occurrences of the regular expressions. Whenever it finds one, it executes the corresponding C code.

The `flex` input file consists of three sections: *definitions*, *rules* and *user code*. The *definitions* section contains declarations of simple *name* definitions to simplify the scanner specification, and declarations of *start conditions*, a mechanism for conditionally activating rules. The *rules* section of the `flex` input contains a series of rules of the form:

```
pattern    action
```

The patterns in the input are written using an extended set of regular expressions. Each pattern in a rule has a corresponding action, which can be any arbitrary C statement. Finally, the *user code* section is used for companion routines which call or are called by the scanner.

At runtime, the generated scanner analyses its input looking for strings which match any of its patterns. Once the match is determined, the action corresponding to the matched pattern is executed, and then the remaining input is scanned for another match.

[2]Paragraph 3.2

Token Streams

For our application, it is not possible to predict the start of a malicious code before processing begins. Thus, every token must be searched for at every byte offset to have complete intrusion detection. When a token is detected at a given byte offset, the scanner will insert its offset to the output stream regardless of other tokens that might overlap the pattern. Since no two consecutive tokens from scanner input should overlap each other, the output must be reformed into one or more valid token streams.

A specific attack scheme can often embed its payload at more than one location within a packet. Therefore, the scanner has to look for tokens at every byte alignment. Furthermore, the scanner maybe looking for several starting tokens for grammars representing different classes of attacks.

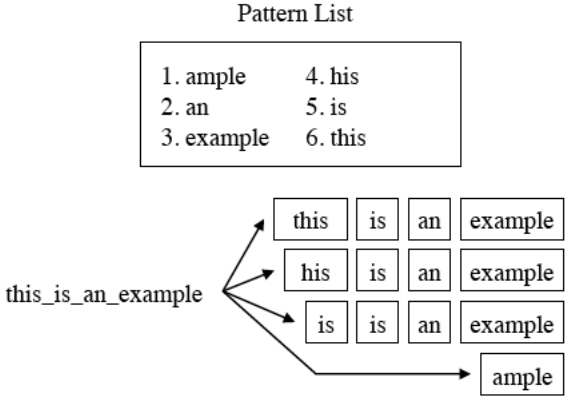

Figure 4.2: Multiple token streams from a single data stream.

Figure 4.2 is an example of how one input byte stream maybe properly recognized as four independent token streams. If we knew where the code started, as with compilers, only one of the four streams would be of interest. Since the code of the attack may be located anywhere in the payload, all four streams must be considered viable threats. Therefore we have modified our pattern scanner to produce multiple streams.

4.2.2 Automatic parser generator `Bison` [6]

`Bison` generates optimized parsers for LALR(1) grammars. In these grammars, it must be possible to tell how to parse any portion of an input string with just a single token of lookahead. That is, parsers for LALR(1) grammars are deterministic, meaning that the next grammar rule to apply at any point in the input is uniquely determined by the preceding input and a fixed, finite portion (called a lookahead) of the remaining input.
A context – free grammar can be ambiguous, meaning that there are multiple ways to apply the grammar rules to get the same inputs. Unambiguous grammars can also be non – deterministic, meaning that no fixed lookahead always suffices to determine the next grammar rule to apply. With the proper declarations, `Bison` is also able to parse these more general context – free grammars, using a technique known as GLR parsing (for Generalized LR). `Bison`'s GLR parsers are able to handle any context – free grammar for which the number of possible parses of any given string is finite.

A `Bison` grammar file has four main sections: *prologue*, *declarations*, *rules* and *epilogue*. The *prologue* section contains macro definitions and declarations of functions and variables that are used in the actions in the grammar rules. The *declarations* section contains declarations that define terminal and nonterminal symbols, specify precedence, and so on. The *rules* section contains one or more Bison grammar rules. A Bison grammar rule has the following general form:

```
result: components...
        ;
```

where *result* is the nonterminal symbol that this rule describes, and *components* are various terminal and nonterminal symbols that are put together by this rule. Finally, the *epilogue* is copied verbatim to the end of the parser file, just as the *prologue* is copied to the beginning.

At runtime, the `Bison` *parser* parses the language described by the grammar we supplied as an input file for `Bison` utility. The job of the Bison parser is to group tokens into groupings according to the grammar rules. As it does this, it runs the actions for the grammar rules it uses. The tokens come from a function called the *lexical analyzer* that is supplied by the `flex` scanner.

Chapter 5

Testing

To validate our new detection engine, we benchmark it against two different signature – based Intrusion Detection Systems. One of the IDSs we use for benchmarking is Snort (engine version 2.8.5.3); Snort is one of the most employed signature – based IDS today. The other IDS we use in the validation phase is PHPIDS which is a state - of - the - art and open – source system for PHP – based web applications. The latter IDS also has an interesting feature: it computes an overall impact value of the attack string it recognizes.

To carry out the test, we employ two different data sets. Because of the lack of publicly – available data sets, containing sufficient data to perform a verifiable benchmark, we were forced to develop our own data sets. Thus, the data sets we benchmark the system against have been developed by us and are based upon different documents publicly available in the Internet.

To carry out our validation we set up a LAMP Web Server. LAMP is an acronym for a solution stack of free, open source software, originally coined from the first letters of Linux (operating system), Apache HTTP Server, MySQL (database software), and PHP, Python or Perl (scripting language), principal components to build a viable general purpose web server. For our purpose we used the following combination of software:

- Debian GNU/Linux 5.0 (lenny)
- Apache 2.0
- php 5.2.6-1
- perl 5.10.0

Our server hosts a web site which has two particular pages that could be exploited for web – based attacks. One of these web pages is a fake login form with usual user – name and password fields. In the other web page a user can submit a message that will be displayed on the page itself.

We focus on HTTP traffic because nowadays Internet attack are mainly directed to web servers and web – based application: Symantec Corporation [4] reports that, in the first half of year 2008, 65% of total discovered vulnerabilities were related to web services and, during the same period, more than 60% of *easily exploitable vulnerabilities* (whenever the exploitation code is not needed or well – known) affected web applications. Symantec also states that typical examples of easily exploitable vulnerabilities are SQL Injection and Cross – Site Scripting (XSS) attacks.

5.1 SQL Injection detection

5.1.1 SQL Injection *signature*

The first part of our signature for SQL Injection is a `flex` input file that describes a scanner. We don't need to set up a complete SQL scanner, we just focus on those most employed keywords used to carry on an attack. Consulting the available literature we selected a set of SQL statements which encompasses all of the statements such as OPEN, CLOSE, and SELECT that actually manipulate a data base[1] and thus could be used to perform a significative attack. This choice leads to the definition of a "partial grammar" for SQL.

An extract of the *rules* section of this partial scanner is shown by Program 2 and Program 3. The action corresponding to each pattern, is focused on returning a token value for the next step of signature elaboration (e.g. parsing). However we can take advantage of this part of the rule definition to make the scanner do something more. Thus, any time we find a "dangerous" token, we update a global counter called `tokno`. We will exploit this counter to compute a simple *threat – score* as we introduced in paragraph **??**.

The second part of the signature is a `Bison` grammar file that describes the language we want to parse. This file mainly consists in a set of grammar rules whose task is to construct each nonterminal symbol from its parts.

[1] Of course this selection is due to the fact that we are using MySQL Server. For MSSQL Server we shod use some different keyword

These rules are defined by a BNF – like syntax. Program 4 and Program 5 show an extract of the grammar rules we employ in our signature.

5.1.2 SQL Injection data set

To test our system on SQL Injection attacks, we set up a data set composed of five SQL Injection attempts and five legal requests. The five attack attempts are shown in Table 5.1.

ID	Username	Password
1	`x'OR'1'='1`	`x'OR'1'='1`
2	`x'OR'2'>'1`	`x'OR'2'>'1`
3	`x'OR'attack'<'threat`	`x'OR'attack'<'threat`
4	`x'OR'1'/*comm`	`ents*/='1`
5	`UNION /*comments*/ SELECT * FROM [...]`	

Table 5.1: SQL Injection attempts thrown against our dedicated server.

We used two types of signature – evasion technique. The first type is the well known "`OR 1=1`" technique: patterns 1, 2 and 3 are all variations of this widely used type of attack. The second technique consists in taking a C – like comment into the attack string. The aim of the latter technique is to evade those signatures attempting to detect a SQL keyword followed, or preceded, by any amount of white spaces. In most cases the C – like syntax for comments can replace any of the spaces. Pattern 4 of our data set is a mixed form of the techniques described above. Pattern 5 is completely based on the second technique; this string can be inserted directly in the url request or maybe in one of the available input field once we got access to the database with one of the previous techniques. Finally, in order to check for false positive alerts, we inserted five other requests that a qualified user would have insert to login, i.e. allowed username and password. Thus, we sent a total amount of ten requests.

Snort has a total amount of twelve rules for SQL Injection attempts detection, they are divided between two files: eight signatures are located in `sql.rules`[2] and the other four rules are located in `web-misc.rules`[3].

[2]Lines: 113, 125, 126, 130, 131, 135, 139 and 146
[3]Lines: 337, 495, 512 and 513

All the rules used by PHPIDS are collected into the same XML file, called `default_filters.xml`. PHPIDS has a total amount of eighteen rules for SQL Injection attempts detection.

Figure 5.2 summarizes the results we obtained at the end of the test. We submitted a total amount of ten requests. Snort did not detect any of the attack attempts, none of the twelve SQL – Injection – related rules matched any submitted pattern. PHPIDS detected all of the attack attempts; more precisely, pattern 1 as well as pattern 2 matched four rules, on pattern 3 we had five matches, on pattern 4 six matches and again six matches on pattern 5. Our system too detected all of the attack attempts reaching a detection rate of 100%.
False positive rate is 0% for all of the three systems, meaning that no false alerts were triggered.

Rates	CoDE	Snort	PHPIDS
DR	100%	0%	100%
FP	0 (0%)	0 (0%)	0 (0%)

Table 5.2: Comparison between our system and Snort using our own crafted data set; DR stands for Detection Rate (attack instance percentage), while FP is the False Positive Rate (packets and corresponding percentage).

5.2 Cross – Site Scripting

5.2.1 Cross – Site Scripting *signature*

The first part of our signature for SQL Injection is a `flex` input file that describes a scanner. In the same way we did for SQL Injection detection, we don't need to set up a complete script language scanner, we just focus on those most employed keywords used to carry on an attack. Consulting the available literature we selected the following set of keywords, including Javascript statement as well as HTML statements: `script`, `alert`, `img`, `src`, `onerror`, `javascript`, `iframe`, `eval`, `location`, `object`. The selection of the previous statements results in a "mixed – language scanner".

The definition of the scanner is shown by Program 6. In this part of the signature, we take advantage of `flex` *start condition* mechanism. Any

ID	Attack Pattern
1	`<script>alert(document.cookie);</script>`
2	`<script language="JavaScript">alert('document.cookie');</script>`
3	`<img src=1.png onerror=javascript:alert(document.cookie);>`
4	`<FRAMESET><FRAME SRC=\"javascript:alert('document.cookie');\"></FRAMESET>`
5	`<script>document.write("Here your cookie:"+unescape(document.cookie));</script>`

Table 5.3: Cross – Site scripting attempts thrown against our dedicated server.

rule whose pattern is prefixed with "<SC>" will only be active when the scanner is in the start condition "SC". Start conditions are managed within a stack structure and three routines are available for manipulating the stack. These routines correspond to the three operations available for any stack structure, that is: push, pop and top. This mechanism allow us to fine tune the computation of the *threat score*, that is we can assign different scores to the same token depending on the context we find that token in.

The second part of the signature is a `Bison` input file that describes a parser for the token stream coming from the scanner we described above in this section. This file mainly consists in a set of grammar rules whose task is to construct each nonterminal symbol from its parts. These rules are defined by a BNF – like syntax. Program 7 and Program 8 show the grammar rules we employ in our signature.

5.2.2 Cross – Site Scripting data set

To test our system on Cross – Site scripting attacks, we set up a data set composed of five legal requests and five XSS attempts. The five XSS attempts are quite different and they are shown in Table 5.3.

We built up the other, non – threatening, requests in order to simulate normal traffic that "accidently" contains some keywords that could trigger an alert. For example a message like the one in Example 1 should not trigger any alerts even if it contains the `</script>` tag.

A Cross – Site scripting attack could be attempted using the <script> tag followed by some javascript code and the </script> tag.

Example 1: A message containing keywords that could trigger an alert.

Although most of the signature – evasion techniques, for XSS attacks, are based on the hex encoding of the attack payload, our data set is only composed of plain – text patterns. The reason of encoding is that to hide the request an unaware user is involuntarily sending. Thus, encoding is useless for our purpose because we do know the requests we are sending and we just want to test our system in order to see whether it will detect those selected attacks. However, Snort, the IDS we are benchmarking our system against, is not detecting all of the attacks we are throwing, even if they are not encoded.

Snort has a total amount of thirteen rules for XSS attempts detection. All rules are collected into `web-misc.rules` file[4]. PHPIDS has a total amount of forty rules for XSS attempts detection. These rules are also collected in the same XML file we mentioned above for SQL Injection rules.

Table 5.4 summarizes the results we obtained at the end of the test. As well as in the previous test, we submitted a total amount of ten requests. Snort detected attack 1 and attack 5 reaching a detection rate of 40%, one of the thirteen XSS – related rules matched both patterns. Snort also triggered one false alert, meaning that one of the non – threatening request we sent contained some tokens that matched one or more rules. PHPIDS detected all of the attack attempts, pattern 1 matched 6 rules, pattern 2 and 3 matched ten rules then we had 9 matches on pattern 4 and pattern 5. PHPIDS also triggered 2 false alerts. Our system too reached a detection rate of 100% but it triggered no false alerts.

Rates	CoDE	Snort	PHPIDS
DR	100%	40%	100%
FP	0 (0%)	1 (10%)	2 (20%)

Table 5.4: Comparison between our system and Snort using our own crafted data set; DR stands for Detection Rate (attack instance percentage), while FP is the False Positive Rate (packets and corresponding percentage).

5.3 Performances

Snort uses the Boyer–Moore pattern–matching algorithm when attempting content matching on the packet payload [5]. This pattern-matching algorithm

[4]Lines: 26, 246, 251, 252, 468, 469, 488, 491, 492, 496, 499, 505 and 506

is one of the most efficient algorithms for string matching and is often used for the search and/or replace commands within a text editor. The worst-case to find all occurrences in a sequence needs approximately 3*N comparisons, hence the complexity is O(N), where N is the length of the sequence, regardless whether the text contains a match or not. The Boyer–Moore algorithm is good for a single string search, but when dealing with a NIDS a single packet can partially match many different rules and for each rule the algorithm will have to be run.

As we stated before, regular – expressions are a nice and powerful formalism, but they can not describe all languages thus they have some limits in the context we are interested in. Any language that can be described by a regular expression is called a regular language. To go beyond regular expressions, we looked at context – free grammars. Context – free grammars are a generalization of regular expressions. Using context – free grammars we can describe more languages.

Given a sequence (e.g. the packet payload) and a context – free grammar, it is possible to decide if the sequence is described by the grammar in $O(N^3)$ time and $O(N^2)$ space. This isnt very good, so tools that are based on context – free grammars usually restrict the grammars they accept so as to allow strategies that result in O(N) time and O(N) space. These restrictions restrict the set of languages the tools can handle; this is rarely a problem in practice, since people tend to create languages that can be parsed quickly. `Bison` is a popular tool for generating C programs that dissect a file according to a given (restricted) context – free grammar.

Chapter 6

Conclusions

We have presented a new approach for signature – based network intrusion detection by developing a system which, taking advantage of high – level – language processing technologies, performs advanced network packet inspection.

The most effective way to protect a computer network against intrusions, is at present represented by Intrusion Detection and Prevention Systems. Unlike firewalls, these systems search for specific patterns in all parts of the packets, increasing the packet inspection effectiveness. However, most of the current IDS implementations do not extend beyond recognizing a set of predefined regular expressions and that makes IDSs prone to false positive alerts and false negatives. Although these pattern matching filters can be powerful tools for finding suspicious packets in the network, they are not capable of detecting other higher – level characteristics that are commonly found in malware.

In Chapter 1, we introduced some motivations for which modern IDSs are to be considered leaking in performances. We hypothesize that IDS' main limitation is that they cannot check for semantics in attack patterns and this is because regular – expressions – based filters do not have the possibility (i.e. they are not so powerful) to detect the context they are working in. That is, if a pattern is matched, according to a certain regular – expression, we can not state whether that match is only a coincidence or it is a real threat, and this leads to false negatives alerts and to false positives. Thus, our system is based on the idea that the ability to detect high level features of the pattern, like the language structure, can lead to more accurate and advanced forms of filters, improving intrusion detection capabilities and drastically reducing

false positive alerts.

In order to catch those high level features, we took advantage of high – level – language processing technologies. As we stated in Chapter 3, language recognition process can be divided into two parts: analysis and synthesis. Analysis is mainly focused on verifying and constructing software structure using grammatical rules, while synthesis is responsible for detecting semantic errors and checking type usage. The analysis phase can be ulteriorly divided into two steps: lexical analysis, or scanning, and syntax analysis, or parsing. As we are not interested in semantic errors nor in code generation, we focused our efforts on understanding how to adapt the analysis phase of language – processing process for network packet inspection purposes.

As well as compilers, which use scanners and parsers to analyze computer program source codes in order to determine correctness in the language structure, our systems exploits scanners and parser to effectively recognize language structures within network traffic. During lexical analysis, our system reads the stream of characters making up the pattern and converts the characters into predefined sequences called tokens. Then, syntax analysis uses the tokens produced during lexical analysis to create a syntax tree representing the grammatical structure of the token stream. The rules describing which tokens are allowed and how to set those tokens up in the syntax tree are defined by a grammar; thus grammars are the signatures of our system. Using grammars as signatures allow us to catch an entire class of attacks (i.e. all or most of the attack variations). Our system architecture is described in detail in Chapter 4.

To validate our system we benchmarked it against two well known and widely used IDSs: Snort and PHPIDS. Since nowadays Internet attacks are mainly directed to web servers and to web – based applications, we focused our tests on HTTP traffic and particularly on two typical examples of easily exploitable vulnerabilities (i.e. whenever exploitation code is not needed or well – known): SQL Injection and Cross – Site Scripting. In Chapter 2 we have presented some examples of the most employed techniques to perform those attacks, of which regular – expressions – based signatures maybe used to detect the attack patterns and how those signatures could be evaded. Thus, we developed one signature for each SQL Injection and Cross – Site Scripting attacks, by defining two restricted grammars. SQL Injection signature is a partial grammar for SQL which describes only those manipulative statements mostly used in attacks patterns. Cross – Site Scripting signature is a mixed grammar which encompasses a meaningful subset of both HTML

and Javascript keywords. The signatures we employed in the testing phase are described in Chapter 5 and in Appendix B.

To carry out the tests, we set up one data set for each class of attack. Each data set is composed by an equal amount of attack patterns and innocuous patterns. The results we obtained are summarized in Chapter 5. Our system totally satisfied our initial hypothesis, reaching a detection rate of 100% and a false positives rate of 0% on both SQL Injection and Cross – Site Scripting data sets. PHPIDS also reached the same detection rate of our system on both data sets but it triggered some false alerts while checking for Cross – Site Scripting attempts. Snort had worse results with respect to both PHPIDS and our system, detecting only 40% of Cross – Site Scripting attempts and none of SQL Injection attempts.

Relying on the results we obtained on our tests, we can state that Snort default signatures are quite ineffective, mostly those concerning SQL Injection. We believe that the main reason for this unacceptable ineffectiveness is to be found in the *specificity* of Snort signatures. Snort default rules are not as much general as it would be required, meaning that they focus on a specific variant of an attack and they are not taking in account all other variants. This make Snort signatures very easy to evade. Furthermore, in previous versions of Snort detection engine, a ruleset file named `web-attacks.rules` was available, whose aim was to detect common attack patterns which exploit web applications vulnerabilities, however this file is no more available in latest Snort version. The reason for this fact is explained by Snort developers team itself:

> *These signatures are going away. They were based on generic signatures that would catch common commands issued to exploit form variable vulnerabilities, but it is so trivial to evade these that it gives the user a false sense of security. Plus, many of them were wrong.*

This "guiltiness admission" confirms what we stated before about the motivations for which Snort signatures are ineffective.

PHPIDS get rid of this problem by augmenting the number of signatures for each class of attack. In this way the system detection capabilities are certainly improved but the cost of this improvement is an augmented CPU consumption and false positives rate. Furthermore, PHPIDS is PHP – dependent thus it can only be employed in PHP based web application.

Thus, we can state that our approach to intrusion detection is an effective alternative to the state–of–the–art intrusion detection systems in terms of detection rate, false positive rate, performances and portability. Of course some improvements have to be done. Future works will focus on a better tuning of the *threat score*, that we just introduced as a possible feature of our system, on the capability of processing encoded patterns, which is one of the most used signature – evasion technique and maybe on the employment of the system in professional or enterprise environments.

Appendices

Appendix A

Overview on Modern IDSs

A.1 Snort

Snort is a signature – based IDS, it uses a set of rules (*signatures*) to check for errant packets in the monitored network [1]. Snort can carry out several tasks, for example it can analyze network traffic, getting packet off the wire and matching them against a set of well – known attack signatures, or it can verify protocols used by packets in order to check for traffic anomalies. Snort has the ability to notice port scan activities, which usually precede an attack attempt. Moreover, each one of the possible threats identified by these control tasks, can be signaled in many ways.

The latest Snort implementation is Snort 3.0

A.1.1 Snort 3.0

Snort 3 Architecture consists of two main component:

- Snort Security Platform 3.0 or SnortSP
- Detection Engines

SnortSP is designed to perform essentially as an "operating system" for packet – based network security applications. It provides common functionalities that all programs that deal with packets need such as configuration load-

[1] As mentioned before a rule, or signature, is a set of requirements that would trigger an alert.

ing, logging and event generation, data acquisition, decoding and validation, flow management and so on. This is the core of the Snort 3.0 architecture.

The second major component are "engines" which is really more of a term to define the analysis modules that plug into SnortSP. SnortSP is designed to be able to run one or more engines, passing them traffic, processing their output and even taking action upon the traffic streams in real – time.

SnortSP

Data Source The Data Source component encapsulates common functionalities required by any network traffic analyzer, functions that will have to be performed prior to running almost any analysis task. The data source incorporates a number of components including:

- Data Acquisition (DAQ) — The DAQ provides an interface between the rest of the engine and the host OS packet facilities. This is where we get packets from the underlying hardware and where we talk to that hardware regarding the disposition of those packets. The DAQ subsystem allows Snort 3.0 to incorporate arbitrary external packet interfaces including things like libpcap, IPQ and divert sockets.

- Decoder — Validates the packets, detects protocol anomalies and provide a referential structure for the rest of the program to operate upon.

- Flow Manager — The Flow Manager provides services for tracking conversations between endpoints on the network. Snort 3.0 also includes a mechanism called "flow slots", that other subsystems can use to store stateful "flow local" information.

- IP Defragmenter — This module provides services for defragmenting IPv4 and IPv6 datagrams and includes mechanisms to allow target – based fragment reassembly.

- TCP Stream Reassembler — As with the IP Defragmenter, provides target-based services for reassembling TCP segments into normalized streams and presenting them to the underlying analytics.

- Data Source API — An abstraction API between the facilities provided by the data source and the rest of the Snort 3.0 software framework.

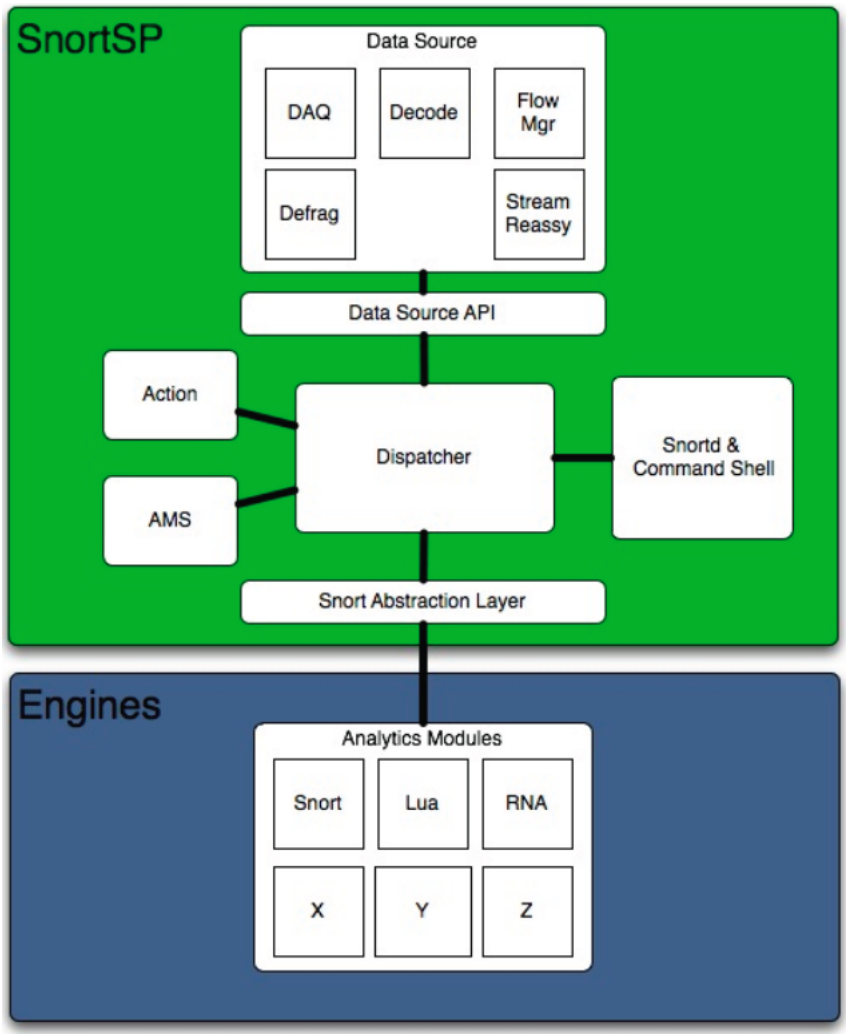

Figure A.1: Snort 3 Architecture

Action System The Action System handles event queuing, notification and logging when the system fires events. The supported output types in Snort 3.0 are text (console), syslog and Unified 2, a serialized binary stream format.

Attribute Management System (AMS) The AMS will store network contextual data about the operational environment being defended by a particular Snort instance. This subsystem will be addressable continuously at runtime and provide interactive interfaces to the command shell as well as analytics modules that can leverage its data.

Dispatcher The Dispatcher coordinates information flow between different components of Snort 3.0 and manages traffic queuing and disposition across analytics threads. It also ties together all of the objects in a runtime instance of a Snort engine, uniting data source, analytics, action system and the attribute manager into a single manageable entity for the purposes of process and threat management from the command shell.

Snortd and the command shell Snortd is the daemon process that provides marshaling services for the objects that are instantiated in a particular framework instance. The command shell also runs attached within a thread to snortd. The command shell provides interactive object management services for different software modules, runtime management of the process and threads, health management and a full scripting language for Snort 3.0. The command shell is running the *Lua*[2] scripting language, a lightweight embeddable scripting language that is fast and portable as well as being very nice for implementing Domain Specific Languages.

[2]Lua is designed, implemented, and maintained by a team at PUC – Rio, the Pontifical Catholic University of Rio de Janeiro in Brazil. Lua was born and raised at Tecgraf, the Computer Graphics Technology Group of PUC – Rio, and is now housed at Lablua. Both Tecgraf and Lablua are laboratories of the Department of Computer Science of PUC – Rio. "Lua" means "Moon" in Portuguese. As such, it is neither an acronym nor an abbreviation, but a noun. More specifically, "Lua" is a name, the name of the Earth's moon and the name of the language.

Detection Engines

Analytics System The Analytics System is where the Snort detection engine threads are located. The idea in Snort 3.0 is to put all detection logic in analytics modules that run as separate threads, all the other code exists to support the functions of the Analytics System. Multiple threads may operate on the data coming from a dispatcher instance simultaneously. The Analytics System is structured so that all interactions between the analytics modules and the rest of the Snort 3.0 framework is brokered by an API called the "Snort Abstraction Layer" (SAL).

A.2 PHPIDS

PHPIDS (PHP – Intrusion Detection System) is a simple to use, well structured, fast and state–of–the–art security layer for PHP based web applications. The IDS neither strips, sanitizes nor filters any malicious input, it simply recognizes when an attacker tries to break the web site and reacts in exactly the web – site administrator wants it to. Based on a set of approved and heavily tested filter rules any attack is given a numerical impact rating which makes it easy to decide what kind of action should follow the hacking attempt. This could range from simple logging to sending out an emergency mail to the development team, displaying a warning message for the attacker or even ending the user's session. PHPIDS enables the user to see who's attacking his site and how and all without the employment of log files or searching hacker forums for user's domain. PHPIDS is licensed under the LGPL.

Currently the PHPIDS detects all sorts of XSS, SQL Injection, header injection, directory traversal, RFE/LFI, DoS and LDAP attacks. Through special conversion algorithms the PHPIDS is even able to detect heavily obfuscated attacks this covers several charsets like UTF-7, entities of all forms such as JavaScript Unicode, decimal- and hex-entities as well as comment obfuscation, obfuscation through concatenation, shell code and many other variants.

PHPIDS requires at least PHP 5.1.6 to use all its features. Depending on which kind of logging and caching the users choses he might need a database that is able to work together with PDO. SimpleXML is required if the user wishes to use the XML based filter rules, as an alternative he can use the fallback JSON based rules. A nice to have for the generic attack detection is Unicode support for the PCRE engine.

Users can configure all important values in the Config/Config.ini. The settings should work out of the box most times but then and when a user might want to tweak the paths or change the way of how the PHPIDS uses caching.

The impact value indicates the severity of the attack. The PHPIDS brings around fifty filter rules to detect attacks and each one of them has an impact, the more rules match on the incoming data, the more likely it's an attack and the higher ranks the resulting impact.

Users can store the impact as session value, if they want to track an attackers activity for some time and wish to react later, for example when

session impact has risen to 50 or 100. A usual very first attack impact is around 5 – 10, sometimes 15 – 20. A typical XSS probing monitored by session based impact usually results in an impact of 50 – 150. So its pretty easy to separate the false alerts from the real attacks using session based impact value.

The PHPIDS is being developed under constant profiling with xdebug and performance measurements to make sure that web applications will not become noticeably slower. Only request parameters are checked whose values inhabits characters besides a–Z, 0–9, @ and _. Furthermore, the performance hungry components are only included and used in case there is input coming in with a key matching the ones given in the Config.ini, thus normally they won't be loaded during about 95% of all requests.

Appendix B

Programs examples

B.1 SQL Injection signature extract

Program 2 Rules for SQL Injection scanner

```
ALL           { return ALL; }
AVG           { return AMMSC; }
MIN           { return AMMSC; }
MAX           { return AMMSC; }
SUM           { return AMMSC; }
COUNT         { return AMMSC; }
AND           { return AND; }
ANY           { return ANY; }
BETWEEN       { return BETWEEN; }
BY            {return BY; }
CHAR(ACTER)? { return CHARACTER; }
CLOSE         { ++tokno; return CLOSE; }
COMMIT        { ++tokno; return COMMIT; }
CONTINUE       { return CONTINUE; }
DECIMAL       { return DECIMAL; }
DELETE        { ++tokno; return DELETE; }
DISTINCT       { return DISTINCT; }
DOUBLE        { return DOUBLE; }
DROP          { return DROP; }
```

Program 3 Rules for SQL Injection scanner

```
ESCAPE       { return ESCAPE; }
EXISTS       { return EXISTS; }
FETCH        { ++tokno; return FETCH; }
FLOAT        { return FLOAT; }
FOUND        { return FOUND; }
FROM         { ++tokno; return FROM; }
GO[ \t]*TO   { return GOTO; }
GROUP        { return GROUP; }
HAVING       { return HAVING; }
IN           { return IN; }
INDICATOR    { return INDICATOR; }
INSERT       { ++tokno; return INSERT; }
INT(EGER)?   { return INTEGER; }
INTO          { ++tokno; return INTO; }
IS           { return IS; }
LIKE          { return LIKE; }
NOT          { return NOT; }
NULL         { return NULLX; }
NUMERIC      { return NUMERIC; }
OPEN         { ++tokno; return OPEN; }
OR           { return OR; }
PRECISION    { return PRECISION; }
REAL          { return REAL; }
ROLLBACK     { ++tokno; return ROLLBACK; }
SELECT       { ++tokno; return SELECT; }
SET          { return SET; }
SMALLINT     { return SMALLINT; }
SOME         { return SOME; }
UNION        { ++tokno; return UNION; }
UPDATE       { ++tokno; return UPDATE; }
USER         { return USER; }
VALUES       { ++tokno; return VALUES; }
WHENEVER     { return WHENEVER; }
WHERE        { ++tokno; return WHERE; }
WORK         { return WORK; }
```

Program 4 Rules for SQL Injection parser

```
sql_list:
sql';'
| sql_list sql';'
;

/* Partial SQL language */
sql: manipulative_statement
;
manipulative_statement:
close_statement
| commit_statement
| delete_statement_searched
| fetch_statement
| insert_statement
| open_statement
| rollback_statement
| select_statement
| update_statement_searched
| query_spec
;
close_statement:
CLOSE cursor
;
commit_statement:
COMMIT WORK
;
delete_statement_searched:
DELETE FROM table opt_where_clause
;
fetch_statement:
FETCH cursor INTO target_commalist
;
insert_statement:
INSERT INTO table opt_column_commalist values_or_query_spec
;
```

Program 5 Rules for SQL Injection parser

```
open_statement:
OPEN cursor
;
rollback_statement:
ROLLBACK WORK
;
select_statement:
SELECT opt_all_distinct selection
INTO target_commalist
table_exp
;
[...]
update_statement_searched:
UPDATE table SET assignment_commalist opt_where_clause
;
query_spec:
SELECT opt_all_distinct selection table_exp
;
```

B.2 Cross – Site – Scripting signature extract

Program 6 Rules for Cross – Site Scripting scanner

```
"script" {
    if (BEGIN(S)) {
        ++tokno;
        return SCRIPT;
    }
}
[<]?"img"   {
    if (BEGIN(I)) {
        return IMG;
    }
}
"iframe"   {return IFRAME;}
"eval"   {
    return EVAL;
}
"location"   {return LOCATION;}
<S,I>src=   {return SRC;}
<S,I>"javascript"   {
    ++tokno;
    return SCRIPT_TYPE;
}
<S,I>alert   {
    ++tokno;
    return ALERT;
}
<I>onerror {return ONERROR;}
```

Program 7 Rules for Cross – Site Scripting scanner (1)

```
script:
script_pure |
script_html
;

script_pure:
SCRIPT alert_stmt |
SCRIPT src_stmt
;

simple_par:
INTNUM |
NAME
;

script_html:
img_stmt |
iframe_stmt |
obj_stmt |
loc_stmt |
eval_stmt
;

input_par:
'('INTNUM')'|
'('NAME')'
;

img_stmt:
IMG src_stmt |
IMG src_stmt onerror_clause
;

iframe_stmt:
IFRAME src_stmt
;
```

Program 8 Rules for Cross – Site Scripting scanner (2)

```
obj_stmt:
OBJECT src_stmt
;

src_stmt:
SRC embedded_script |
SRC simple_par |
SRC input_par
;

onerror_clause:
ONERROR alert_stmt |
ONERROR ASSIGN alert_stmt |
ONERROR ASSIGN eval_stmt
;

embedded_script:
SCRIPT_TYPE ':' alert_stmt
;

alert_stmt:
ALERT input_par |
ALERT simple_par
;

eval_stmt:
EVAL input_par |
EVAL '('src_stmt')'|
EVAL '('alert_stmt')'
;

loc_stmt:
LOCATION embedded_script |
LOCATION simple_par
;
```

Bibliography

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Thechniques, and Tools*. Addison Wesley, 2 edition, August 31, 2006.

[2] Jay Beale, James C. Foster, Jeffery Posluns, and Brian Caswell. *Snort 2.0 Intrusion Detection*. Syngress Publishing, 2003.

[3] Damiano Bolzoni. *Revisiting Anomaly - Based Network Intrusion Detection Systems*. PhD thesis, University of Twente, Distributed and Embedded Security Group, 2009.

[4] Symantec Corporation. Internet security threat report 2008, 2008. `http://www.symantec.com/enterprise/threatreport/index.jsp`.

[5] Neil Desai. Increasing performance in high speed nids, a look at snort's internals. *http://packetstorm.linuxsecurity.net/papers/IDs/*, 2002.

[6] Charles Donnelly and Richard Stallman. *Bison*. 2.4.1 edition, 2009.

[7] C. Kruegel and T. Toth. Using decision trees to improve signature - based intrusion detection. *RAID'03: Proc. 6th Symposium on RecentAdvances in Intrusion Detection*, Volume 2820 of LNCS:Pages 173–191, 2003.

[8] Vern Paxson, Will Estes, and John Millaway. *Flex manual*. 2.5.35 edition, 2007.

[9] Wikipedia. Intrusion detection system, 2009. `http://en.wikipedia.org/wiki/Intrusion-detection\_system`.