Linguistic Security Testing for Text Communication Protocols

Ben W. Y. Kam, Thomas R. Dean

School of Computing, Queen's University, Kingston, Canada
Electrical and Computer Engineering, Queen's University, Kingston, Canada

Abstract. We introduce a new Syntax-based Security Testing (SST) framework that uses a protocol specification to perform security testing on text-based communication protocols. A protocol specification of a particular text-based protocol under-tested represents its syntactic grammar and static constraints. The specification is used to generate test cases by mutating valid messages, breaking the syntactic and constraints of the protocol. The framework is demonstrated using a toy Web application and the open source application KOrganizer.

Keywords: security testing, mutation testing, text-based communication protocol.

1.Introduction

Despite widespread knowledge of classes of security bugs [16, 19], vulnerabilities continue to occur. Security faults have serious consequences, such as the theft of information or the complete failure of the system. This paper describes a framework for testing that applies transformation techniques from the program comprehension literature to generating test cases specific to the security of the system. Our general approach is similar to previous research on binary protocols [1, 17, 18, 22], but the flexibility of text based protocols such as iCalendar [6] or HTTP [8] raises new challenges.

In our approach, we describe the protocol using a context free grammar with XML markup to specify additional lexical, syntactic and context sensitive constraints. From this augmented grammar we automatically generate a markup engine that transfers the markup to captured valid test data. The markup is used to mutate the test data to check for security vulnerabilities. We demonstrate the framework against applications using the HTTP and iCalendar protocols, discovering a previously unknown vulnerability in the Qt library in the process.

In next section, we discuss the goals of this paper. SST framework overviews and SST components anatomy will be illustrated in Section 3 and 4 respectively. Section 5 states the SST low/middle levels concrete architectures. Section 6 reports experiments in SST and follows with the related work. Finally, the conclusion and future work will be drawn in the last section.

2.Goals and History

Binary protocols such as OSPF [15] are protocols in which the data exchanged is transmitted in a similar representation to that used in memory. For example, the integer value 4 is transmitted as the binary value 0x04 (8 bits) or the value 0x00000004 (32 bits). In text based protocols such as HTTP, use ASCII or UNICODE, and the value 4 may be transmitted as the ASCII character ‘4’ 0x34. While binary protocols provide some flexibility in lengths of fields, the number and order of fields in the messages is fixed. Syntactic mutations to messages such as deleting a field have little meaning, as the next sequence bytes in the message will be interpreted by the system under test as the new value for the field. Binary protocols also tend have limited support for the nesting of structures. Text based protocols have a flexible syntax, often allowing extra spaces and newline characters, and when MIME [4] or XML [5] are used as part of the encoding, allow flexible ordering and deletion of fields. Thus the syntax and lexical properties of the protocol become valid concerns for security and robustness testing.

Our previous versions of Protocol Tester [1, 17, 18, 22] handled binary protocols by translating them to a textual form, mutating them using program transformation techniques and then translating back to the binary format. The protocols were described using a context dependent grammar, and XML markup that specified constraints such as the types of fields or the relation between the length of one field and the value of another. These markups are used by a test planner to insert a different set of XML markup tags into the captured message sequences to guide the mutation. While the tags used to guide the mutation was flexible and expandable, the set of tags available for use in the protocol specification was hard coded into the tool set, requiring code modification when they were extended.

Thus the goal of SST was a lightweight framework capable of handling the more complex mutations for text protocols and at the same time supporting an easily extensible markup system for specifying constraints in the protocol description. As specified by Beizer “data validation is the first line of defense against a hostile world”, all input data should conform to its grammar and the best input format should be defined as a formal language [3].

3.SST Framework Overviews

The SST framework is similar to the structure of Protocol Tester, and consists of a total of five modules: Capture, Markup, Mutate, Replay, and Oracle. Fig. 1 shows the five components of the SST framework.

The protocol dependent module **Capture** is responsible for capturing and decoding the network traffic between the client and the server. Capturing is done by a sniffing component (Sniffer), which in the case of web applications is a modified version of the Firefox browser allowing us to capture encrypted messages (https) in unencrypted form. If the captured response messages are compressed or encoded a decoding component is be invoked translate them to plain text. The Capture module also creates a manifest file. The manifest file specifies the protocol, the server addresses, port numbers and the information of proxy servers for each message. This allows SST to test systems spanning multiple servers.

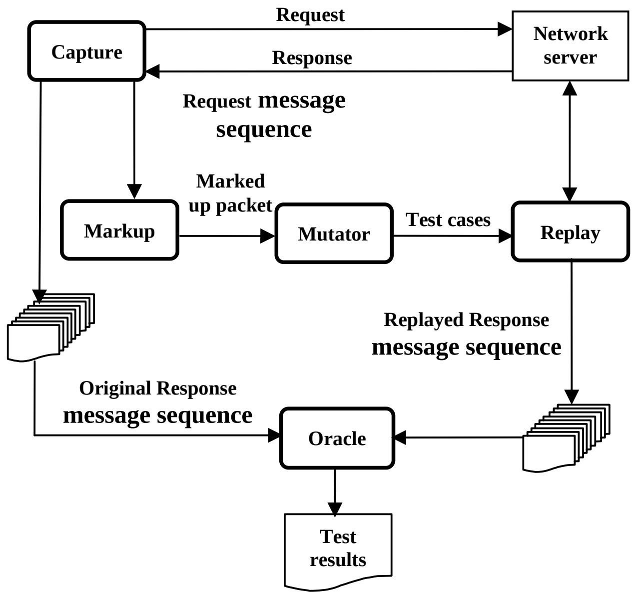

Figure 1: SST overview

The **Markup** module uses the protocol description file to insert markup into the captured messages. This markup is then used by the **Mutator** module to generate the test cases. Both of these modules are protocol independent.

The **Replay** module uses the manifest file generated by the capture module to transmit the test cases to the server(s). When the mutated message is not the first message in a sequence, the original versions of the previous messages in the sequence are sent. The current version of the module is largely protocol independent, with a custom component handling HTTP cookies and session information. In the future this will be made protocol independent by adapting the approach for specifying state dependent messages used in Protocol Tester [22].

4.Protocol Specification and Markup

SST uses a protocol specification to mutate captured messages to generate the test cases. We distinguish between three levels of protocols in the specification. At the lowest level we have the base format of the captured messages. Since we are interested in text protocols, this is the lowest level above the TCP/IP stream protocol, such as the HTTP protocol. This lowest level protocol may also serve as a container for other protocols. For example, SOAP [9] can be used to encode remote procedure calls within the HTTP protocol. At the highest level we have the application protocol which assigns application specific meaning to messages, such as the messages related to shopping carts. In this paper we discuss the specification of the low and middle level protocols.

```
% partial HTTP grammar
define program
    [request-message]
end define
define request-message
    [request-line][repeat headers_message]
    [CRLF][opt message_body]
end define
define request-line
    [method][space][request-uri][space]
    [http-version][CRLF]
end define
```

Figure 2: The partial low level HTTP protocol

```
Include "http.grm"
redefine entity_header
      …
    | [SOAPAction]
end redefine
define SOAPAction
    [soap_uri][soap_message]
end define
define soap_message
    [xml_declaration][open_soap_envelope]
    [soap_header] [soap_body][close_soap_envelope]
end define
```

Figure 3: The middle level XML SOAP protocol specification

4.1 Syntax Specification

The protocol specification is created based on the syntax specification of the protocol. We use the TXL language to specify the syntax of the protocols. Figure 2 shows a partial grammar for the HTTP protocol. The non-terminal `program` specifies the goal symbol of the grammar. Square brackets are used to indicate the use of another non-terminal, the keyword `repeat` indicates multiple instances of a non-terminal and the keyword `opt` indicates 0 or 1 instance. So in the figure, a `request_message` consists of a `request_line`, followed by multiple `headers_message`, a `CRLF` and an optional `message_body`.

Middle level protocols are specified by extending the lower level protocols. Foremaple in figure 3, the SOAP protocol is defined by first including the HTTP grammar (the `include` statement) and then extending the entity_header non-terminal (the `redefine` statement). The `entity_header` non-terminal was previously defined in the HTTP grammar.

4.2 Grammar Markup

The syntax of the protocol is extended using XML markup to specify constraints. In the SST framework, the meaning of these constraints are open ended, as they are simply markers to signal the location where the mutators should operate on the messages. SST also supports the specification of linked tags. That is, a markup tag that can be used to specify a relationship between two separate elements of a message.

The grammar is used to place the markup tags at the appropriate locations in the captured messages. To specify the use of markup, the tester places the XML tags in the grammar surrounding the grammar elements that represent the sections of the message that the tester wishes to mutate. Figure 4 shows an example. In this example, the request-line definition has been marked with both the `enumeratedLiteral` and `caseSensitive` tags. These indicate that the method of the request line is one of a limited set of literal values, and is case sensitive. When multiple tags are used,

```
define request_line
    <enumeratedLiteral>< caseSensitive >[method]</ caseSensitive ></enumeratedLiteral>
        [space]  [request_uri]  [space] [http_version] [CRLF]
end define
```

Figure 4: Markup tags in the protocol grammar.

they must be properly nested. From the grammar, SST generates a program that inserts the markup into the appropriate place in the captured messages.

Figure 5 shows a snippet of the result of running the generated insert markup program against a captured HTTP post request message. The request line has been wrapped in the figure, but in the marked up message, it is a single line. As can be seen from the figure, the XML markup has been inserted surrounding the literal `POST` which is matched by the `method` non-terminal.

Figure 6 shows an example of a relationship tag. Relationship tags are identified by presence of the `id` and `root` attributes. In this case all of the markup with the same tag value are considered related to each other in some way. In this particular case we are indicating that the value given in the `Content-Length` mime header gives the length of the message body. Unlike the similar constraint in Protocol Tester, this tag is not used as part of the parsing process, but used to indicate the relationship so that the length mutator may make appropriate changes. Since the grammar may match more than one instance in a given message, the `id` attribute is used to identify each instance that was recognized. The % character is replace with a unique integer as each instance is matched. The `role` attribute is simply copied allowing the mutator to identify which part of the message is represented in each markup.

Figure 7 shows the instantiation of the length tag from figure 6 in a captured message. The length tag with the length role has been added to the `Content-Length` header, while the length tag with the `value` role has been added to the message body. There is no limit to the number of roles for a markup tag that can be specified, all will be inserted into the captured message by the generated markup program.

```
<enumeratedLiteral><caseSensitive>POST</caseSensitive></enumeratedLiteral>/return.asp
HTTP/1.1
Host: 192.168.1.105
…
```

Figure 5: Nested Markups on the method POST

```
define Content_Length
   'Content-Length : [space] <length id="%" root="request_message" role="length"> [number]
     </length>
end define
define message_body
   <length id="%" root="request_message" role="value">
   [repeat token_or_key]
   </length>
end define
```

Figure 6: Length linked tag in the grammar

```
…
Content-Length: <length id="1" root="request_message" role="length">48</length>
…
<length id="1" root="request_message" role="value">FirstName=John&LastName=Smit
h &DOB=10%2F15%2F1980</length>
```

Figure 7: Length linked tag in captured message

4.3 Markup tags

As mentioned in the last section, each markup is implemented by its own mutator. The generated insert markup engine simply moves the markup from the grammar to appropriate parts of the captured messages. Thus the set of mutator tags is entirely open ended. We demonstrate the framework with an initial set of markup tags and mutators that illustrate the different purposes they serve and the types of mutators that can be created.

Table 1 shows these initial markup tags for which mutators have been created. The first of these, the `enumeratedLiteral` tag illustrates a tag in which the mutator is generated from the grammar specification. It is used to indicate that the purpose of the non-terminal is to generate one of a list of literal values. While this can be inferred from an analysis of the grammar, the use of the tag allows the tester to indicate which of these non-terminals should be tested. A separate program analyzes the grammar, and for each instance of the `enumeratedLiteral` tag, genererates a mutator that will alternate the values based on the values given in the grammar. In the example in Figure 4, the `method` non-terminal was marked with this tag. The `method` non-terminal recognizes the set of HTTP methods: `GET`, `POST`, `OPTIONS`, `HEAD`, `PUT`, `DELETE`, `TRACE` and `CONNECT`. The generated mutator will modify the method in the message shown in figure 5 from `POST` to each of the other alternatives. Similar mutators can be generated based on common syntax vulnerabilities such as missing termination tags.

Lexical tags are used when the lexical constraints are stricter than the lexical tokens used in the grammar, or we want to substitute particular values for the tokens. Our initial set of tags deals with changes to the case of the token, changes to individual characters (for example, substituting "," and ":" for "." in the HTTP version of the request line), deletion of arbitrary literals such as mime headers, and

Table 1. The categorization of markup tags

Types	Tags	Purpose
Syntactic	enumeratedLiteral	Change to another terminal provided from grammar to alter the original semantics
Lexical	caseSensitive	Change the terminal letters from upper case to lower case or vice
	charSpecific	Change the terminal character
	dateSpecific	Change the terminal date format
	syntaxSpecific	Alter the terminal characters
	valueLimitation	Change the terminal value to common boundary values
	stringSpecific	Replace a string values with common alternate strings
Relational	length	Indicates that the number marked by the length role gives the number of characters in the value role.
Custom	jpeg	The content identified by the tag is an embedded jpeg image (e.g. file upload).

changing values of integers and strings. The current mutators for integer and string values targets buffer overflows, but other mutations are easily introduced.

We have only implemented one relationship tag, the `length` tag, but another candidate tags is a mime type tag that links the Content-type header to the message body allowing mutators to recognize specific content types for mutation. We have implemented one custom tag that is inserted when embedded jpeg image are recognized (image gallery web applications, for example). In this case, the mutator extracts the embedded jpeg image, invokes an external binary mutator and then inserts the resulting mutated images back into the request messages.

The markup can be specified by the tester in one of two ways, it can be manually inserted directly into the protocol grammar, or alternatively, it can be specified separately from the grammar. Fig 8 shows the use of the Grammar Merge Program that merges a markup specification into a Generalized Protocol Grammar. The markup specification contains alternate versions of grammar definitions from the generalized grammar that includes the markups. It may also contain additional definitions that are used in the alternate grammar definitions. Fig 9 shows an example of such a file. The example shows a definition of `http_version` that parses the version number as two numbers separated by a period, and the period has been annotated with the charSpecific tag. The original definition of http_version, might use a single floating point number. Thus this approach allows us to write a more general protocol grammar and then specialize it for alternate testing strategies. In particular, when crafting a grammar for a new protocol, we could use agile parsing techniques[7] such as robust parsing and island grammars to adopt a minimal grammar specification and then extend each part of the grammar in separate markup files to be tested independently. Figure 10 shows an example of such an approach for the

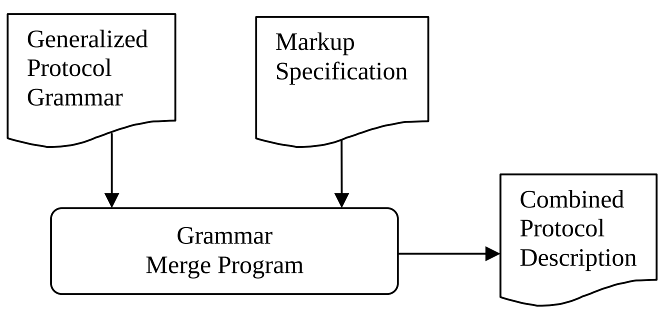

Figure 8. The categorization of markup tags

define http_version
HTTP / [number] <charSpecific> [period] </charSpecific> [number]
end define

define period
'.
end define

Figure 9: A markup specification file

```
define http_version
   [repeat not_CRLF_Token_or_Key]
end define

define not_CRLF_Token_or_Key
   [not CRLF] [Token_or_Key]
end define

define Token_or_Key
   [token] | [key]
end define
```

Figure 10: Generalized grammar

`http_version` non-terminal. In this variation, the http version is any sequence of tokens or keywords that is not a carriage return followed by a linefeed. The `not` keyword in TXL means that the particular non-terminal cannot be parsed at this point in the input.

This approach has several advantages. First there is no need to implement the grammar for the entire protocol, only the portions which are to be tested. Second, if the generalized grammar is written exactly to the protocol specification, parts may be difficult to mark. Thus the markup specification can provide alternate parses making the markup tag placement easier. It also allows several testers to operate in parallel, each using separate markup specifications on different parts of the generalized grammar. Lastly, it is difficult to get a generalized grammar that will be suitable for all testing. The markup specification can modify the grammar appropriately for each test.

Figure 11 shows the process diagram of this portion of SST. The combined protocol description is used to generate an insert markup program. The insert markup program in turn is used to parse and insert markup into each of the captured messages. The marked messages are then passed to mutators which run independently to produce the test cases.

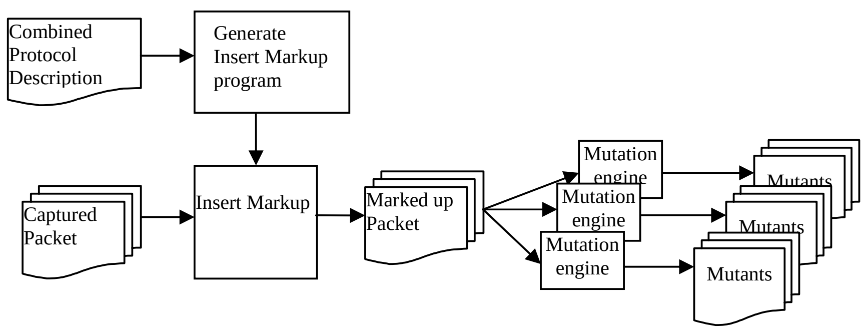

Figure 11: Markup and mutate process

5.Replay and Oracle

The replay module consists of four components, an injector, a travel agent, a realtime update module and a decoder. The process flow of this component is shown in Fig 12. The injector is the primary component responsible for overall control of the replay process. It fetches each of the mutants from the test suite, and uses the travel agent to send the test case to the server. The travel agent is responsible for communicating with the server. It handles monitoring the connection for the response, and handling timeouts if the server crashes. The real-time update component is used if the protocol has state dependent elements. For example, some web applications use session cookies, or encode session identifiers into the URLs.

The realtime update component monitors the response messages and modifies the appropriate elements of the request messages. The current real-time update component is protocol independent using a regular expression matching engine to locate the elements in the response and request messages. However the program that generates the configuration file for this component is HTTP specific. In the future, the approach can be made protocol independent by adopting the approach used by Zang et al.[22].

The Injector is also responsible for maintaining the state of the database on the test server. If needed, the injector will reinitialize the database, typically restoring it from a snapshot prepared for the test.

The decoder component handles any compression or encoding of the response packets, storing the response sequence in clear text so that the Oracle can compare against the original set of responses.

The current oracle contains two phases addressing this task. The first phase is to check whether the injector has completed each test run. This means all the packets in a test run have been sent to the server. In some situations, the injector will stop the test run after the mutated packet has been sent. This may be because the server is unable to respond to any more requests after receiving the mutated packet. If the test run passes the preliminary check, then the oracle will start a detailed analysis.

A detailed analysis is the second phase and consists of two stages. The first one compares each character of the original response message to the response message received from the mutated request message. If they are identical, it means the

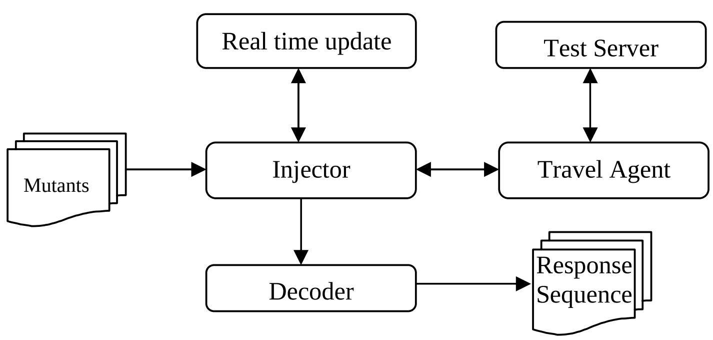

Figure 12: Replay process structure

response message received from the mutated request message is well-formed. However, if they are not identical, the current oracle cannot make the verdict that the response message received from the mutated request message is well-formed. The oracle will generate the report and the tester needs to analyze this report to make the final decision.

6. Experiments

We have tested our approach by conducting two experiments on two separate protocols. The first experiment is designed to show the correctness of the SST framework. The second experiment on the iCalendar protocol demonstrates the protocol independence of SST, and exposes a new vulnerability in the open source application kOrganizer.

6.1 Toy Web Applications

Several toy web applications were constructed that contained vulnerabilities for six tags. These servers were used to validate the functionality of the framework before attempting to find new, unknown errors in other applications. A total of six small tests were conducted to demonstrate six different mutated packets that were sent to the toy server successfully causing a web application, database and/or web server to run with anomalous behavior.

One of the test cases uses the `caseSensitive` tag to change the method of the request message “POST” to “post”. In the first test, IIS accepted the request message and stored the posted message to the database. This experiment was retested using the Apache2 server. The mutated message was correctly rejected by the Apache2 server. All of the planted vulnerabilities were discovered by the framework.

6.2 kOrganizer

The second experiment applied the framework to the open source kOrganizer. In this case the capture module was not a modified firefox browser, but iCalendar files generated by kOrganizer and Apple’s iCal. Instead of using an injector as the replay component, we use a xmacroplay [21] to script the opening of the mutated iCalendar files (kOrganizer cannot be given an iCalendar file on the command line). The oracle is also simple since we are looking for a catastrophic failure of kOrganizer (i.e. kOrganizer crashes). Thus our test is a script that copies each mutated iCalendar file to a specified directory, opens a new instance of kOrganizer and runs an xmacroplay script to instruct kOrganizer to open the file and exit. If the exit status is abnormal, or the kOrganizer process is still running (i.e. it has deadlocked), then an error is reported. There was an inherent inefficiency as xmacroplay must include multiple worst case delays to ensure that the appropriate dialog box has been rendered by kOrganizer before a mouse or keyboard event is sent.

In this experiment, the caseSensitive, charSpecific, dateSpecific, syntaxSpecific, valueLimitation, and the stringSpecific tags were used to generate a total of 1026 test cases from a single iCalendar file. The total running time was 244188 seconds (67.83 hours). Table 2 and Table 3 show the experimental setup information and testing data, respectively. Of the 1026 test cases, one error was logged (test case 559). This test case was one of those generated by the stringSpecific mutator to insert multiple string values. This particular case changed the description field to a 16 Megabyte string, causing a segmentation violation (SIGSEGV). Examining the code revealed that the vulnerability was actually in the Qt interface library used to build KDE applications.

Table 2. The Second experimental setup information

Computer	Operating system	Memory	KOrganizer
AMD3300+	Ubuntu 8.10	512M	4.1.4

Table 3. The data of the experiment two

Create 1086 Mutants	9.069s
Remove tags	45.742s
Test driver runtime	244188s
Total	244242.811s

7.RELATED WORK

There are many security flaws that can be found in literature about web applications security testing. These flaws are created by violating the fundamental of CIA security requirements. CIA stands for confidentiality, integrity, and availability. Confidentiality holds when only authorized users have the ability to access data. Integrity ensures data cannot be altered by an unauthorized user. Availability requires that data should always be available to legitimate users.

If the CIA security requirements of web application is not met, multiple consequences can result. First, it is possible to cause the web application, database, and/or web server to crash. Second, users' data and/or system information can be stolen and/or modified. Third, computer resources can be wasted by illegal users. Table 2 shows different kinds of security flaws caused by breaking CIA security requirements. A slight change in the content of the packet by breaking the syntax and/or semantics of the grammar will break the CIA. SST provides markup tags to instruct mutation engines explicitly to perform the changes. For example, stringSpecific tag instructs the mutation engine to replace the original string value with a specially crafted string for SQL injection. If the attack is successful, the information could be altered and/or stolen and compromises the confidentiality (C) and/or integrity (I) of the security requirements.

TABLE 2. Consequence of CIA security requirements violation

CIA security requirements violation	Security flaws
Confidentiality	Information stolen
Confidentiality	Information alternation
Confidentiality	Privacy violations
Confidentiality	Impersonation
Integrity	Web application crash
Integrity	Web server crash
Integrity	Database crash
Integrity	Information alternation
Availability	Wasting computer resources
Availability	Take over the system

There is a great deal of research on security testing of web application. Much of this research focuses on SQL injection, cross-site scripting and command injection. Some research also provides method to generate guards in the applications from the models. User input strings must be passed through the guards for security checking prior to accessing the database. Jing et al [12] use a non-deterministic finite state machine to mutate packets. However their approach, like our previous research is focused on binary protocols. Text is more flexible and less susceptible to value changes.

Aitel's block-based network protocols security testing [2] is the most similar to SST. However, the test cases generation obtained by random fuzzing variables only breaks the syntactic constraint of the protocol grammar. SST, in addition to generating random fuzzing values, also provides different types of markup tags to violate syntactic and semantic of the protocol grammar. For example, relational type markup tags break the semantics relationship between terminals.

Guido et al [13, 20] use the relational calculus and automata to formal model the system's required security requirements. Their security testing only can test application level security. SST not only can test application level security, but also low level and middle communication protocol level security.

Halfond et al [10, 11] propose a combination of static and dynamic analysis for SQL injection protection of web application. For the static part, they build models based on static analysis of the source code that contains all of the possible legitimate SQL queries in a PHP application. Test cases generation is accomplished by injecting additional SQL statements into a query to intentionally violate the model. The dynamic analysis incorporates the comparison of runtime queries with the static model. If the dynamic query violates the model, execution is halted and noted.

Merlo et al [14] use dynamic analysis of legitimate test cases and security scenarios to build static models corresponding to the call site. A query invocation at the call site will be compared to the corresponding model to check whether or not it is a legitimate query. Both approaches focus on SQL injection, and do not address cross-site scripting or command injection. The approach of Merlo et al has the advantage of not relying on the source code and thus is capable of testing SQL-injection by malicious code. Their method can be reused for other languages but an execution environment with appropriate instrumentation is required.

The main contribution of our approach is that it is protocol independent, and can be used to test most text based protocols. The other contribution is that we can easily generate tests for multiple vulnerabilities such as cross-site script injection and command injection. In addition our approach requires only the specification of the protocol and direct access to the database to detect modification of the application data.

8.Conclusion and Future Work

SST is a lightweight framework for generating security and robustness test cases for text based network applications. The protocol is easily expanded by adding markup tags and the mutators to implement each of the tags. We have demonstrated the framework by expressing and testing two applications, each of which uses a different protocol.

There are several ways in which the system can be extended. The first to add more markup tags and more mutators. There is a lot of inherent flexibility in the system. The current mutators only use attributes in the markup for the relationship type of tag. The only attributes that have special meaning are the id, root and role attributes. Other attributes can be used to pass parameters to the mutators such as range of numeric values or expected maximum lengths for strings. In addition mutators need not only use single tag types, but may perform mutations on multiple tags simultaneously.

The second avenue of exploration is more syntactic dependent mutators. The only one implemented in the SST prototype was enumeratedLiteral. One extension is to identify and insert markup for non-terminals that implement this role by analyzing the grammar. Other grammar based markup can also be added, such as changing the order of or deleting elements of the captured messages.

We have already extended SST to handle higher level application protocols that are build on top of the lower level protocols such as HTTP and SOAP. This involves a domain specific language to automatically generate recognizers to identify messages that are syntactically similar but semantically different, such as the difference between a login request and a list shopping cart request. This allows the grammar to be refined using agile parsing techniques and the tester to insert markup tailored to the semantics of the message such as mutating user names and passwords in login requests.

REFERENCES

1. AboElFotoh, M., Dean, T.R., Mayor, R, “An Empirical Study of a Language Based Security Testing Technique”, Proc. 19th IBM Centres for Advanced Studies Conference, Toronto, Canada, November 2009, pp 112-121.
2. Aitel., D., “The Advantages of Block-Based Protocol Analysis for Security Testing”, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.116.1178& rep=rep1&type=pdf, last accessed 2010.
3. Beizer B., Software testing techniques, Van Nostrand Reinhold Company, New York, 1990, ISBN: 0-442-24592-0.
4. Borenstein, N., Freed, N., “MIME Part One: Format of Internet Message Bodies,” Internet RFC 2045, 1996.
5. Bray, T, Paoli, J, Sperberg-McQueen, C.M., Maler, E, Yergeau, F, “Extensible Markup Language (XML) 1.0 (Fifth Edition)”, W3C , 2008.
6. Dawson F., Stenerson D., “Internet Calendaring and Scheduling Core Object Specification(iCalendar),” Lotus, Microsoft, IETF RFC 2445, 1998.
7. Dean T. R., Cordy J. R., Malton A. J., Schneider K. A., “Agile Parsing in TXL”, Journal of Automated Software Engineering, pp. 311 – 336, 2003.
8. Fielding R., Irvine UC, Gettys J., Mogul J., Frystyk H., Masinter L., Leach P., Berners-Lee T., “Hypertext Transfer Protocol - HTTP/1.1,” Compaq/W3C, Compaq, W3C/MIT, Xerox, Microsoft, W3C/MIT, IETF RFC 2616, 1999.
9. Gudgin, M, Hadley, M, Mendelsohn, N, Moreau, J, Frystyk, H, Karmarkar, A, Lafon, Y, “SOAP Version 1.2 Part 1: Messaging Framework (Second Edition),” W3C Recommendation, 2007.
10. Halfond W. G. J., Orso A., “AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks”, Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering (ASE '05), pp. 174-183, 2005.
11. Halfond W. G. J., Orso A., “Combining static analysis and runtime monitoring to counter SQL-injection attacks”. In Proceedings of the 3rd International ICSE Workshop on Dynamic Analysis (WODA), IEEE Computer Society Press, pp. 105-110, 2005.

12. Jing C., Wang Z., Shi X., Yin X., Wu J., “Mutation Testing of Protocol Messages Based on Extended TTCN-3”, Proceedings of the 22nd International Conference on Advanced Information Networking and Applications, pp. 667 – 674, 2008.
13. Jurjens J., Wimmel G “Formally Testing Fail-safety of Electronic Purse Protocols”, Automated Software Engineering, IEEE Computer Society, pp. 408 – 411, 2001.
14. Merlo E., Letarte D., Antoniol G., “Automated Protection of PHP Applications Against SQL-injection Attacks”, Proceedings of the 11th European Conference on Software Maintenance and Reengineering, pp.191-202, 2007.
15. Moy, J, “OSPF Version 2”, Internet RFC 2328, 1998.
16. OWASP, The Open Web Application Security Project, http://www.owasp.org/, last accessed: 2009.
17. Tal O., Knight S., Dean T. R., “Syntax-based Vulnerabilities Testing of Frame-based Network Protocols”, Proceedings of the Second Annual Conference on Privacy, Security and Trust, 2004
18. Turcotte Y., Oded T., Knight S., Dean T. R., “Security Vulnerabilities Assessment of the X.509 Protocol by Syntax-Based Testing”, Proceedings of MILCOM 04 on Military Communications Conference, pp. 1572 – 1578, 2004.
19. WASC Projects. “Web Application Security Consortium, Threat Classification.” http://projects.webappsec.org/ Threat-Classification/, last accessed: 2008.
20. Wimmel G., Jurjens J., “Specification-based Test Generation for Security-Critical Systems Using Mutations”, ICFEM, LNCS, Springer-Verlag, pp. 471-482, 2002.
21. Xmacro, http://xmacro.sourceforge.net/, last accessed April 29, 2010
22. Zhang S., Dean T.R., Knight G.S., “Lightweight State Based Mutation Testing for Security”, Proc TAICPART-MUTATION 2007 , Windsor, UK, pp. 223–232, 2007.