Analysis of HTTP Protocol Implementation in Smart Card Embedded Web Server

Nassima KAMEL* and Jean-Louis LANET*
*Smart Secure Devices (SSD) Team, University of Limoges, France

ABSTRACT

The latest generation of smart card embeds an HTTP web server which facilitates the integration of smart card into the existing networks and provides more services and custom interfaces. It also helps the developers to simplify the use of new programming model (servlets). However, due to the sensitive information stored and the resource constraints with which the technology is running, it is necessary to test it deeply. Our aim is to detect bugs and vulnerabilities and non-compliance of the HTTP embedded web server. For that purpose, we used the fuzzing technique which consists of injecting invalid or random data on various inputs of the software to be tested. Our fuzzing tool, Smart-Fuzz is based on the Peach framework customised to our needs. Moreover, working in black box, we created the PyHAT application to collect maximum information of the target features. Thus, we can reduce the amount of protocol functionalities to be analysed. The results generated in the log files are finally analyzed to understand the behaviour of the application and to detect if some fuzzed data has succeeded to take up the vulnerabilities.

Corresponding Author:
Nassima KAMEL
Smart Secure Devices (SSD) Team, XLIM Labs, University of Limoges, France,
123 rue Albert Thomas, 87000, Limoges, France.
Email: nassima.kamel@unilim.fr

1. INTRODUCTION

The integration of a web server on smart card is a technology promoted by Open Mobile Alliance (OMA) which provides the Smart Card Web Server (SCWS) specification. The SCWS is built on the HTTP protocol version 1.1 for the smart cards dedicated to mobile phones which is based on Java Card 2.2 platform and for low-end devices. Moreover, Oracle (formerly Sun MicroSystems) proposed the Java Card 3 connected edition platform (JC3) in which an HTTP embedded web server is defined with improved features through the enhancement of the framework with newly supported Java API and programming model. However, it is used for high-end platforms.

A web server usage exposes the card to threats that could affect the platform security. For this reason, we need to guarantee that no vulnerability is present in the card. There are two ways to eliminate vulnerabilities:

- Build error free software with a formal method and generate automatically the code from the specification [16],
- Generate test suites to verify if some particular security properties are respected in the implementation under test.

Often formal model checks only if the specification is correct, while with test approach the implementation is checked. The smart card is a highly constrained device and the HTTP protocol is a complex and error-prone protocol. As a result we choose to elaborate a strategy to give information on the implementation. Our work aims at testing the compliance and the robustness of the HTTP protocol implementation, particularly in the embedded web servers. Thus, we have chosen the fuzzing technique for its effectiveness in auditing different types of applications and systems. The paper [3] presented the use of this technique in the verification of compliance of the the transport layer protocol available on a smart card with the specification defined by ETSI (European Telecommunications Standards Institute) [10].

We have no knowledge of the target implementation as we are working on a black box model. We have developed the PyHAT application to collect as much as possible information about the smart card (detection of:
implemented HTTP methods, supported versions, supported encoding methods, etc.). This step is very important since it reduces the number of tests to be performed at the fuzzing step. We also developed our fuzzer program, Smart-Fuzz, written in Python language which provides Data and State Models, corresponding to methods and headers detected by PyHAT. Smart-Fuzz is based on Peach framework, which is an open-source framework that allows to develop fuzzer tools adapted to some environments.

In this paper, we first introduce the architecture of the SCWS and the embedded HTTP protocol and subsequently the state of the art of the protocol validation and the fuzzing technique. Then, we describe our contribution and explain the main points of our tool, namely: the PyHAT strategy to discover the supported features of the web server, and Smart-Fuzz steps to generate the tested data and the logging interpretation. And finally, we present some experimental results and conclude our work.

2. SCWS ARCHITECTURE

2.1. SCWS Card Web Server

The SCWS is a standard HTTP/1.1 [11] web server dedicated to smart cards and specified by the Open Mobile Alliance (OMA). It facilitates the integration of smart cards into the existing networks and eases the card administration over standard protocols. The SCWS is both a server, and a client application. In the server mode, it is used by the subscriber via HTTP client (Web browser). In client mode, a card issuer may remotely administrate the SCWS with an OTA (Over The Air)[2, 8]. To communicate with the handset, an SCWS gateway is needed when the SIM card does not support a TCP/IP stack (Figure 1). The local transport protocol will mostly be the Bearer Independent Protocol (BIP) [7, 10]. A SCWS Gateway ensures the translation of the data format between the SCWS and the HTTP client (Web browser). It translates the HTTP requests from the client to local protocol specific commands (BIP) and the server responses to HTTP responses. Also, it allows the enforcement of an access control policy (ACP).

A Bearer Independent Protocol (BIP) channel (BIP) [7, 10] allows the SCWS to be run independently from the card issuer network. To communicate with the mobile, the card uses BIP commands (Figure 1):

- **OpenChannel**: open a communication channel between the card and the mobile
- **GetChannelStatus**: ask to the mobile the communication channel state
- **SendData**: send data to the mobile
- **ReceiveData**: receive data from the mobile
- **CloseChannel**: close a communication channel

To communicate with the card, the mobile uses two events:

- **DataAvailable**: informs the card that the mobile wants to send data
- **ChannelStatus**: asks to the card the communication channel state

These commands are composed of one or more Tag-Length-Value (TLV). They belong to the Subscriber Identity Module (SIM) Application Toolkit (SAT) technology defined by the ETSI (BIP) [10]. The SAT consists of a
A URL format is used to access to the SCWS resources from a browser. OMA defines the following format for URL:

(A) \text{http\_URL} = \text{http://host[:port][abs\_path[?<name>=<value>]]}

(B) \text{https\_URL} = \text{https://host[:port][abs\_path[?<name>=<value>]]}

A SCWS is addressed with the IP address 127.0.0.1. The URL (A) is used for simple HTTP communication, while the (B) provides security over Transport Layer Security (TLS). The specification proposes different port numbers. Using the BIP protocol, the hosting device may implement its own HTTP services, so there is a need for two new port numbers. The Internet Assigned Numbers Authority (IANA) assigned the ports numbers 3516 (smart card Port) for HTTP and 4116 (smart card-TLS) for HTTPS. In case of a SIM card implementing its own TCP/IP stacks (like Java Card 3 connected edition) there is no need for new port numbers. Conventional ports are used (80 for HTTP and 443 for HTTPS).

2.2. HTTP Protocol

The Hypertext Transfer Protocol (HTTP) is a client/server based protocol used in the WWW. The client transmits requests to the server that contains information of the requested resource. The server sends the requested resource (if available) or an error message (BIP) \cite{11}. An HTTP packet is usually encapsulated in TCP packet which is again encapsulated in IP packets. In case of SCWS based smart card that does not support the TCP/IP stacks, the HTTP packets are encapsulated in BIP packets.

**HTTP Request:** An HTTP request contains a sequence of lines, namely:

- The request line which contains the HTTP method, the URI of the requested resource and the used HTTP version. A URI is used to identify the requested resource. The method specifies the request type. The standard HTTP/1.1 defines 11 methods among which 8 of them given below are implemented most oftenly.

  - GET is used to retrieve the resource (in the form of an entity) identified by the Request-URI
  - HEAD is identical to GET, except that the server does not return a body message in the response
  - POST is used to send a significant number of data to a script hosted by the server that will treat them to create or update a resource. The resource is identified by the Request-URI in the request line
  - PUT requests enclosed static resource stored under the supplied Request-URI. This is the only method used for the static resources
  - DELETE method allows to delete the resource identified by the Request-URI field
  - OPTIONS represents a request to obtain information about the communication options available on the request/response chain identified by the Request-URI

- TRACE and CONNECT are supported only by the server applets:

  - TRACE is used to invoke a remote application-layer loop-back of the request message.
  - CONNECT is used with a proxy which may dynamically switch to a tunnel.

- Header field lines which give additional information about the request and / or the client (browser, operating system ...). Each line consists of a name specifying the type of the header and a corresponding value

  
  
  ```
  "field-name": [field-value]
  ```

- Body of the request (the request can optionally contain a body (data)) is mainly used to pass parameters using the POST method. The body is sent after the header lines, it is separated from the last header by an empty line specified by a double CRLF (carriage return and linefeed).
HTTP Response: For each client request, the server returns an HTTP response that consists of a sequence of lines. The first one is always the Status-Line, which consists of the protocol version followed by a numeric status code and its associated textual meaning. The status code has 3-digit number, indicating if the request is successfully executed or not. The other lines are composed of different header fields and potential data.

<table>
<thead>
<tr>
<th>Status code</th>
<th>Textual meaning</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1XX</td>
<td>Informational</td>
<td>indicates a provisional response</td>
</tr>
<tr>
<td>2XX</td>
<td>Successful</td>
<td>indicates that the client’s request is successfully received, understood and accepted</td>
</tr>
<tr>
<td>3XX</td>
<td>Redirection</td>
<td>not supported</td>
</tr>
<tr>
<td>4XX</td>
<td>Client error</td>
<td>returned for cases of error from a client</td>
</tr>
<tr>
<td>5XX</td>
<td>Server error</td>
<td>returned when the SCWS server has an error or it is incapable to perform the request</td>
</tr>
</tbody>
</table>

The implementation of the HTTP protocol on a smart card obligates to some restrictions because in standard implementation multiple buffers are used which is not possible in smart cards with 2 KB of memory. The restrictions on this implementation can cause errors and vulnerabilities, this is why it is necessary to test this platform.

3. STATE OF THE ART

3.1. Protocol validation

Designing a protocol that handles all different probabilities and unexpected cases is difficult. Hence it must taken into account every feasible command and possible sequences of the commands.

For designing robust protocols, several approaches have been proposed like variety of general formalism such as state diagram, Petri nets (BIP) [21], grammar and programming languages [14]. Transition models [12] are often used because protocols are responding to numerous events like commands or timing events. State Machine (SM) modelises a protocol in a natural way with its events forming the SM inputs. Other specification methods that do not require definition of explicit state may also be used. List of input/output history [6] defines the allowed input, output sequences and their relations. Algebraic specifications provide another way of defining the allowed sequence of operations [5].

Once the protocol has been specified we need to verify its implementation. State machines can be used to analyze protocols using a method known as the reachability analysis technique. Using the technique, the global state of the system is expressed for each transition. Each global state is then analyzed. An exhaustive search checks that all reachable states are safe. Reachability analysis is suitable for determining if a protocol is correct with respect to its specification but they do not guarantee to resist the attacks. Another serious limitation of this technique is that drastic assumptions are required in order to keep the number of states small.

The algebraic term rewriting technique is very similar to the SM. It starts with the specification of the initial state. An analysis is then made to show that an insecure state cannot be reached. Although both methods are related, the term rewriting technique is better than conventional SM since it can discover new protocols weaknesses.

All this specification languages and verification methods are based on the model of a protocol and procedure to verify the properties. The main drawback here is that it takes more time to design a model. We have chosen a method called “fuzzing” to automate the model construction with a brute force test generation.

3.2. The fuzzing technique

Fuzzing is a technique which is used to find the errors in software implementations by injecting invalid data [19]. The main goal of fuzzers is to crash the target (machine, application, protocol, etc). An inherent limit of this technique is the time consumption. The main advantage here is to search vulnerabilities with a low-cost material. Fuzzing data can be generated in three different ways [17]:

- Random data generation which has the inconvenience to be blind and not pertinent. In most of the cases the data are filtered and rejected by the target.
- The fuzzer generates an invalid data from a data model created by the user. Then it sends them to the application or protocol to be tested. This method is time consuming because it needs to know integrally the protocol. But it is the most effective method.
Most existent fuzzers are designed to test network protocols such as in [20] that presents a tool to test TCP/IP layer, but their use is not limited to a unique area. This technique was also used for testing protocols implemented on smart cards. In [3] we presented a fuzzer that verifies the compliance of the BIP protocol implementation with specification defined by ETSI. In [15] a fuzzer based on the Sully framework [18] is presented, and dedicated to test and find flaws in smart card secure payment protocols (EMV). Dan Gri [13] proposed a tool called SCFuzz, designed to test Middleware smart cards. By generating exceptions, SCFuzz will allow using the dysfunctions observed in order to discover any exploitable vulnerability. There are many fuzzing frameworks based on APIs that can be used to implement tools for auditing at different levels (files, API, arguments for a command line, standard input, etc.) as:

- SPIKE [1] is one of the first fuzzing frameworks which is written in C and dedicated particulary for developing network protocol fuzzers. It introduces a concept of "blocks"; the data structures are broken and represented as a block that allows abstract construction of various protocol layers with automatic size calculations

- Peach [9] is an advanced and robust fuzzing framework written in Python. It provides an XML file to create data model and state model definition which is used to make data mutations. It is a flexible framework that allows to define our fuzzing strategy (modify one or many data elements at a time, define mutators to use, modify some parts of our model, change the flow of our state model, etc.).

- Sulley [18] is based on SPIKE technique generation of data. It also includes many other important functionalities as managing different state of protocols, using agents to monitor a target (monitoring network communications, detecting eventual faults, VMware Control), and offers some drivers to facilitate the use of Sulley.

The fuzzing technique is used to test the robustness and compliance of the HTTP protocol implemented in the smart card embedded Web server. The tool we have developed is based on the Peach software. Our choice for this software is mainly due to the fact that it has been proven by several studies, especially a work of M.Barreaud [3] that targeted the same environment in which we are interested (SCWS). On the other hand, Peach can perform the tests in parallel on multiple platforms that is very beneficial for us; given the smart card constraints in terms of resources and a limited number of write cycles.

4. CONTRIBUTION
4.1. Overview of our tool

The tool we have implemented allows verifying the implementation of the HTTP protocol in any web server. We are particularly interested on a web server embedded on smart cards. Our tool is based on black box model since we have no knowledge about the targeted server implementation [4].

Our tool has two complementary applications PyHAT (Python HTTP Assessment Test) and Smart-Fuzz (Figure 2). Fuzzing is performed by the Smart-Fuzz which is based on Peach software. The test data are generated using data descriptors (data models, state models and mutators) represeting the HTTP requests and their different constituting fields. The application PyHAT was developed in order to make the fuzzing more intelligent and reduce the execution time due to a large number of tests. PyHAT is performed in the first step where it detects the different supported features by implementing the HTTP protocol in the smart card to be tested. The results of this application
are then used by the Smart-Fuzz tool that limits the generation of tests to only featured implementaion on the target application.

We also applied parallel distributed fuzzing tests on a set of cards to reduce the number of data processed by each card. Fuzzing results are distributed in structured log files which are then automatically analyzed to detect vulnerabilities.

4.2. PyHAT

If a fuzz is done to all existing HTTP methods and their associated headers, a huge amount of tests should be performed. These tests may spend a lot of time especially in case of low bandwidth devices like a smart card. In order to decrease the amount of tests PyHAT application have been developed which can search the implemented HTTP features in the targeted server. As we explained before, the HTTP protocol defines some return values. We will see that these values may be interesting and they may characterize the tested HTTP web server. In the following we presents the functions that pyHAT may detected and the stategy used for each one

This part defines the strategies used for each previous functions and the expected results. As we explained before, the HTTP protocol defines some return values. We will see that these values may be interesting and they may characterize the tested HTTP web server.

Detect implemented HTTP methods: The RFC 2616 [11] describes: ”A server which receives an entity-body with a transfer-coding it does not understand SHOULD return 501 (Unimplemented), and close the connection.”

The strategy chosen to determine the methods implemented in the HTTP server, is based on this response numeric status code. In practice, for each method, a request is defined which respect the RFC 261. In the response, only the first line may be isolated to obtain the return code and to determine if the method is implemented or not (return code is 501).

Detect request case sensitive: The test case sensitive is used to the requested method and fields where the specification does not define any specific rule. To perform these tests, we define a first request in which the method is written in lower-case. In the second step, the method is formatted in accordance with the specification, but other fields are written in an alternating lower and upper letter case. If for each case, the server responds with a code 200 (OK), we deduce that there is no distinction between upper-case and lower-case.

Detect the supported HTTP version: Finding the supported protocol version of an HTTP server is not a friendly task. Indeed, when a request is being sent to an HTTP server it should use the same version (if possible) as that of the client. Our analysis is therefore based on this fact and follows the algorithm as given below:

- Send a request with a specific HTTP version
- Analyze the HTTP version used in the response
- If the HTTP version is same, it is supported by the HTTP server else not supported.

We apply this algorithm for each HTTP version (0.9, 1.0 and 1.1). The RFC 2616 defines a return value (505) if an HTTP version is not supported. But we noticed that some servers do not return an error. That is why we have chosen to perform this test with the previous algorithm which is working for all the cases.

Detect supported encoding data methods: The server must respond to an HTTP request using an encoding managed by the client if it is possible or else with no encoding. Currently the client specifies the encoding which it can accept in a field Accept-Encoding. The server receives the request and analyzes this field. If the value is an encoding mechanism managed by the HTTP server, it will respond with the same encoding indicated in the Content-Encoding field. There are four defined encoding types: gzip, compress, deflate, and identity.

In order to evaluate the encodings supported by the HTTP server, we send one request for each encoding value and we analyze the field Content-Encoding in the response. There are two possible cases:

- There is no encoding field in the server response:
  - The encoding mechanism is not supported
  - The encoding mechanism is implicit and is the identity

- The value of field Content-Encoding is the same as the client request. So the conclusion is that the encoding is managed by the HTTP server

Detect server analysis request fields: In contrast to the previous analysis, the detection of the fields supported by the server is a hard problem. In fact, we do not have any knowledge to implicitly determine if a field is implemented or not.
The strategy chosen here is to send a valid request with a specific field. According to the value of the field, the RFC 2616 describes the expected value from the server. For the fields where there is no defined response in the specification we imagine, by default they are implemented. The table 1 summarizes the strategy used for some fields:

Table 2. Request fields analysis strategies

<table>
<thead>
<tr>
<th>id</th>
<th>Field</th>
<th>Value</th>
<th>Expected code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Accept</td>
<td>text/plain; q=1, text/html; q=0</td>
<td>406</td>
</tr>
<tr>
<td>2</td>
<td>Accept-Charset</td>
<td>ISO-8859-1, utf-8; q=0</td>
<td>406</td>
</tr>
<tr>
<td>3</td>
<td>Expect</td>
<td>10-continue</td>
<td>4xx</td>
</tr>
<tr>
<td>4</td>
<td>If-Modified-Since</td>
<td>Sun, 06 Nov 2020 08:49:37 GMT</td>
<td>200</td>
</tr>
<tr>
<td>5</td>
<td>If-Range</td>
<td>Random UUID</td>
<td>200</td>
</tr>
<tr>
<td>6</td>
<td>If-Unmodified-Since</td>
<td>Sun, 06 Nov 1970 08:49:37 GMT</td>
<td>412</td>
</tr>
<tr>
<td>7</td>
<td>Range</td>
<td>1-30</td>
<td>206</td>
</tr>
<tr>
<td>8</td>
<td>Content-Length</td>
<td>1 000 000 000 000 000 000 000 000 000 000</td>
<td>413</td>
</tr>
</tbody>
</table>

Description: A description is given below for each field listed on the table 1:

1. Forbidden to receive HTML content in favor of “text” content type should cause the server to send a response indicating its inability to provide this type of data (code 406).

2. If the server can not send a response with an encoding requested by the client, it should return the expected response 406.

3. Giving an unknown value of the Expect field should hold the HTTP server to return a 4xx error class value.

4. If the requested resource has not been changed since the specified date, the server must return a code 304. In our case, the date is invalid (2020 is more than the current date). So the server should return a code 200.

5. The value used for the If-range field is randomly generated, it is an improbable match for an entity on the server. So the expected code is 200.

6. When injecting a request that asks for the unmodified resource for a specified time, the server must response with a return code 412.

7. Requesting a partial data should generate the response code 206 if the request is satisfied, else the requested range is not available or this field is not analyzed.

8. If a server receives a request with a content length greater than it can handle, it must return code 413.

Find the resources contained in the web server: To discover all the URIs of a web site, we use the same way as any web site mirroring software. Starting with the index page of the web site, we collect all strings that match to a certain regular expression (in our case, the attributes href). Then we restart the operation on each discovered URI and we do some checks:

- We verify if this URI is new
- If this URI refers to an external resource. We save it only if it has the same IP address of the analyzed HTTP server

The results of PyHAT are stored in an XML file which may be parsed by our fuzzer Smart-Fuzz.

4.3. Smart-Fuzz

Smart-Fuzz is the heart of our HTTP fuzzing tool, it is composed mainly of 4 steps: generating test data, transmission of data to the target, generating log files, analysis of the results (Figure 3). It is designed to test any type of web server (standard or embedded). We are particularly interested on SCWS servers. Its objective is to verify the compliance and robustness of the implemented HTTP protocol by analyzing its behavior when it receives invalid input (crash, unexpected card behavior, behaviors that does not conform to the specification).
Smart-Fuzz is based on Peach software. To generate test data, it uses predefined HTTP requests descriptors and limits for testing only targets supported features which are specified in the PyHAT result file. It is also composed of an optional interface HTTP-BIP, used in case of testing SCWS servers, to ensure the compatibility of the data format with the target.

The vulnerability detection is based on the analysis of returned codes corresponding to the injected data, namely the BIP code, the returned APDU code "status word" which is encapsulated in BIP response and HTTP response. We also applied a parallel fuzzing to optimize the fuzzing time.

4.4. Test generation

Building a representative model of HTTP protocol requires an exhaustive study of the HTTP specification. Based on RFC 2616, the details of the various fields of requests are studied, taking into account the HTTP protocol version. The strategy of our fuzzing tool is to mutate the fields of the injected request and to analyze responses from the server and a card comparing them to the expected responses.

As presented previously, an HTTP request is composed of three levels, a request line, one or more rows for headers and optionally request body:

- **The request line:** The request line is composed of fields: `<Method>`, `<URI>`, `<Version>` and CRLF
  - `<Method>`: It can take the value of one of the method types supported by the platform to test (detected by PyHAT). For example: GET, POST
  - `<URI>`: Sets the absolute or relative path (from the server) that locates the requested resources
  - `<Version>`: Defines the HTTP protocol version that will be used during the communication (used only in the versions above HTTP/0.9). Its syntax is of the form: HTTP / <major>.<minor>, <major> and <minor> are integers

- **HTTP headers:** An HTTP request can be of different fields (header).
  The header allows the client to inform the server about its capabilities (accepted language, encoding, etc.). It must have the name of the header followed by its value, both parts are separated by two points as follows: `<field>: *(<value>)`
  The formats of the header values are different. Having a large amount of information in the HTTP protocol specification, we just take a basic exemple of field here which is the “date and time” field.
  **Example:** Format of date and time
  For compatibility with earlier versions of the protocol, we consider different formats for date and time specified in each standard. Three different formats available are:
  - Sun, November 6, 1994 08: 49: 37 GMT
  - Sunday, 06-Nov-94 08: 49: 37 GMT
  - Sun Nov 6 08: 49: 37 1994

- **<body>:** Is used to route the data defined by the user. It is represented as a random of string characters.

Smart-Fuzz is based on Peach framework that uses XML file named pit-files:
The data model represents the structure of the protocol
The state model defines the basic logic state needed to test the protocol
The publisher describes where the data are sent
The mutator represents the mutation types used

An HTTP request is represented by a set of the fields that constitute it. The fields used are different depending on the request type. A data model is XML representation of the structure of the request fields. It also identifies the elements to be mutated and also in some cases a set of default values. Peach proposed the possibility to distribute the data description by blocks. This feature offers the advantage of reducing the amount of data models to be generated. In fact, each header is represented by its model, and for modeling a request it should simply gather all models corresponding to the headers contained in the request. The Example below represents a data model of the HTTP request which references to another data model named RequestLine that represents the data model of the request line.

Example: Data model of HTTP request

```xml
<DataModel name =" HttpRequest ">
<Block name =" RequestLine " ref =" RequestLine "/>
</DataModel >
```

The mutations to be used are also defined here. Indeed some fields only accept a specific data types such as String. The format of these data can also be a set of values as in the case Method field. Using this approach can reduce considerably the number of generated tests and avoid unnecessary testing with non-compliant data.

Example: Method field mutations

```xml
<choice >
<String value =" GET " mutable =" false " />
<String value =" POST " mutable =" false " />
<String value =" HEAD " mutable =" false " />
<String value =" PUT " mutable =" false " />
<String value =" DELETE " mutable =" false " />
<String value =" CONNECT " mutable =" false " />
<String value =" OPTIONS " mutable =" false " />
</choice >
```

4.4.1. Configuring Peach for SCWS target

From XML files (pit files) that we have defined, Peach generates HTTP data to be injected into the target. However, in the case of SCWS smart cards, based on BIP protocol, the card expects to receive BIP packets encapsulating HTTP packets and Peach does not support the BIP format. To overcome this problem we used an application interface that we call "HTTP-BIP" (Figure 4). This interface acts as an intermediate and it allows to encapsulate HTTP packets in a BIP format. The BIP answers are also handled by the interface HTTP-BIP to recover the HTTP response, which is useful for analysis. We configured Peach to designate the interface HTTP-BIP as fuzzing target (in publisher). We specified a path to the interface that receives fuzzing data that will be transfered to the smart card in BIP format.

4.4.2. Log Files

To verify the presence of faults or abnormal behavior due to non-compliance with the specification of the HTTP protocol, it is necessary to analyze the different messages sent and received by the target server. For this
purpose we save a record of various transactions during fuzzing in event files. There are two ways of generating file events, using the default mechanism of Peach which consists of saving everything in one log file located in the root of Peach, or design a custom log. To facilitate the analysis of the results we defined our own organization of log files.

In the case of SCWS based cards, the logging mechanism is managed by the interface HTTP-BIP. The results are organized by type of HTTP method and header names. For each test, the information about the request and response exchanged between Peach and the target are stored in the corresponding directory. The information retrieved are:

- The couple HTTP request / response
- BIP command
- HTTP return code
- card “status word”

### 4.4.3. Logs Analysis

Analysis of the logs is to look for suspicious or non-compliant behavior. For most of the existing fuzzing tools, this task is performed manually by the user and must verify all the results to detect a defect. To minimize user interaction, we designed an application called Analyzer that automatically checks the presence of predefined patterns in the log files. Suspicious searched behaviors are:

- Requests causing a total shutdown of communications (server crash)
- The absence of HTTP response (the response contains only TCP or BIP packets)
- Code response incompatible with the tested method or header
- HTTP response that do not match with those expected according to the RFC 2616
- Generation of APDU error (Status Word: 6F 00)

The analyzer uses as input for a file containing a set of patterns that it will search in the log files. Among the searched patterns:

- NO CHANNEL DATA (special pattern in the case of SCWS) is a BIP response generated if the smart card responds with no HTTP content
- 6F 00 (special pattern for smart card) is an APDU code which means that the application has thrown an uncaught Java exception
- 5XX: this category of HTTP return codes correspond to an internal server error
- Association of patterns: to check for return codes that do not correspond to a query type, we define regular expressions that consist of a combination of patterns between the request and the response. The application Analyzer searches in log files the presence of one of these predefined regular expressions. For example: the combination of “GET” and “201 Created” patterns is a regular expression. Of course, according to the HTTP specification, return code 201 can not be a response to a GET request because this code means a good sent of a resource while the GET method don’t return data.

Automatic analysis greatly reduces the task to the user who can select a set of faults with our Analyzer application and he can then manually go through the remaining logs for further analysis if necessary.

### 4.5. Experimentations and results

We applied our tool to NFC smart cards embedded web server (SCWS) based on Java Card 2.2 Platform to verify the performance and effectiveness of it. The tested smart cards were provided by a company with whom we have established an agreement that restricts us to keep the information confidential about the provider and also the details of our results. We also tested our tool on a Java Card 3 prototype. The results of our tests have demonstrated the effectiveness of our tool to detect vulnerabilities and non-conformance of the tested HTTP web servers.
4.6. Running time

To calculate the execution time performance of our tool, at first we tested it on a smart card without using the PyHAT application. In this case the fuzzing took more than 8 days and we stopped the execution of fuzzing before it ends. The PyHAT application significantly reduces this time and limits the fuzzing to only methods, headers, versions and encodings implemented in the target. The execution time has been reduced to an average of 6 days.

The execution time is important in testing of smart cards. Therefore we put a parallel fuzzing to reduce the number of tests imposed to each card. The execution time is relative to the number of used cards. We used our tool for three cards parallely. However, one of these cards blocks before finishing the execution of all tests, probably because of the security mechanisms implemented on it. We were unable to identify the exact cause of this case since we did not have any knowledge of the used cards implementation. We noticed that the parallel fuzzing (two cards) reduced the execution time to almost half compared to the fuzzing applied on a single card and the time taken was about 56h.

4.7. Detected vulnerabilities

The smart cards that were tested by us revealed some vulnerabilities and non-compliance with the specification. Among these cases some of them are presented below:

**Permission to add or delete a resource:** we have noticed that some cards accept PUT, and DELETE administration commands. These administrative commands are intended to be used by the card issuer (application administrator) that securely accesses the smart card server to perform tasks such as adding or deleting users, modifying a configuration file, etc. All these tasks should not be accessible by the user who could perform malicious modifications in order to increase its privileges (e.g increasing the number of points on a loyalty application) or attacking other cards. Some tested smart cards revealed flaws in this property.

**Causing a Java exception:** an error in the smart card may generate an error code 6F XX that corresponds to a Java exception. For security reasons, smart cards hide the details of the error by returning for all Java exceptions the code 6F 00. The results showed that the tested SCWS smart cards generate a Java exception (6F 00) after injecting an empty request.

**HTTP response does not conform to the specification:** as previously reported, our tool can detect non-compliance with the specification, when the server’s responses to specified request does not correspond to what is provided in the HTTP specification. Among the results we found that when sending an If-Match header the card responds with an HTTP error code 500, which means that internal server error has occurred, while the card must send a 501 HTTP error (Not implemented) or a 400 HTTP error.

**Incomplete implementation:** the respect for compliance with a specification is also to ensure that all required fields are implemented. The results showed that the tested cards do not correctly implement some rules of the specification. For example:

- The Host field that defines the area of the target application is mandatory in HTTP/1.1 version. The tests we performed by injecting a request without including this required field has no error generated.

- The last HTTP request header and body must be separated by a newline. The results we obtained show that this rule is ignored in the tested SCWS cards. In fact, when sending a request without complying with this rule the card returns the response APDU 90 00 meaning that the transaction was successful but the response does not have HTTP response (no response from the SCWS).

The tests we performed on Java Card 3 smart card revealed that only GET and POST methods were implemented, whereas according to the Java Card 3 specification, the platform is based on the HTTP/1.1 protocol, which defines the GET, POST and HEAD methods which must be implemented. However to see as the first and the only one prototype (to our knowledge), it was very likely that the implementation was not complete.

5. CONCLUSION

The tool that we proposed allows us to audit any HTTP web server, and in particular web servers embedded on smart cards. In this paper we presented certain optimization strategies to make the tool more intelligent and pertinent by defining and implementing a strategy to detect all the supported features, the modeling of HTTP request, analyzing logs based on pattern data base and adapting our tool to the SCWS based smart cards. The various tests demonstrated the effectiveness of our tool to detect defects in the web servers. However, for future works it would be interesting to extend the pattern data base and attempt to exploit detected vulnerabilities for carrying out attacks.
REFERENCES


BIOGRAPHY OF AUTHORS

Nassima KAMEL. Nassima KAMEL received her PhD degree in computer science from the University of Limoges, France in 2012 December. Her thesis was on Security of smart card embedded Web server. She is currently working as a researcher in SSD (Smart Secure Devices) team at Limoges university, France and also a lecturer at University of Limoges. Her research interests include security at application and protocol levels.

Jean-Louis Lanet is a Professor in the Computer Science Department at University of Limoges from 2007. He is also the team leader of SSD (Smart Secure Devices) research group at Limoges university. Prior to that, he was a senior researcher at Gemplus Research Labs (1996-2007). During this period he spent two years at INRIA (Sophia-Antipolis) (2003-2005) as an engineer at DirDRI (Direction des Relations Industrielles) and as a senior research associate in the Everest team. He started his career as a researcher at Elecma, Electronic division of the Snecma, now a part of the Safran group (1984-1995) and his field of research was on jet engine control. His research interests include security of small systems like smart cards and software engineering.