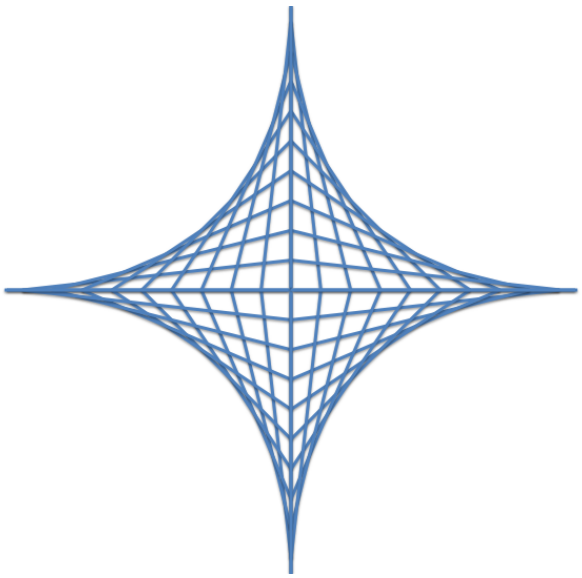
WebSpa
Single HTTP/S Request Authorisation Web Knocking
Specification Guide
_v0.7

Oliver Merki, Yiannis Pavlosoglou

April 21, 2014



seleucus.net

Table of Content

1 Introduction

This document provides a detailed description of the elements that constitute the Hypertext Transfer Protocol (HTTP) Uniform Resource Locator (URL) request issued from the client of WebSpa to the corresponding web server. This introductory section describes where this specification guide fits in the wider remit of WebSpa documentation; it also provides an outline on how the sections of this document are ordered and how they can be read.

1.1 This Guide & Other Supporting Documentation

The discrepant event discussed herein is web knocking. This document is one of three, with the purpose of describing how WebSpa can be used. The three documents are:

- **WebSpa Administration Guide** This document describes how to setup and use the WebSpa server. It details how to create new users and add new action numbers with respective Operating System (O/S) commands assigned to them
- **WebSpa Specification Guide** This document describes the actual design detailing the use case, specification, requirements and actual attacks, which this tool has been engineered to withstand
- **WebSpa User Guide** This document describes how to use the WebSpa client for issuing commands through a URL request to a web server.

The specification guide aims to enable anyone who would be interested in implementing their own version of WebSpa to do so.

1.2 What is WebSpa?

WebSpa is a complete client/server tool that allows you to send premeditated commands to the system your web server is running on.

1.3 Document Structure

The scope of this specification guide is presented in six consecutive sections, with a number of sub-sections. Each section stands as an independent entity and can be read alone. Each section also has a relation to other ones. The following lattice provides the dependancies between individual sections:

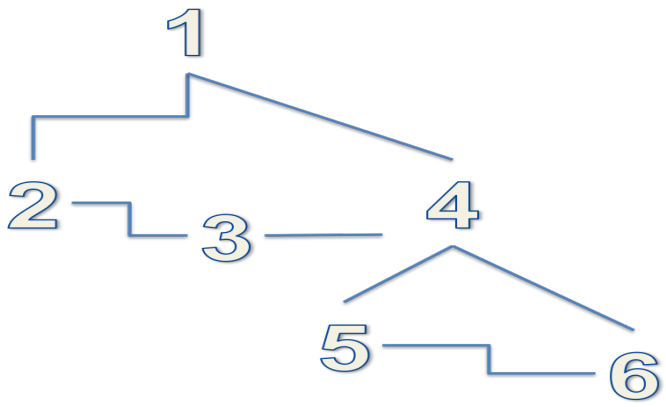


Figure 1: Structure of the document. Looks like a molecule - it's not.

1. **Introduction** - An introductory section, describing the document structure as well as the other type of documentation available
2. **Use Cases** - This section presents in UML the actors and their interactions with WebSpa
3. **Requirements** - Listing the non-functional as well as functional requirements
4. **The Request** - Describes the structure of the request, how it is derived and what is the method of processing this request
5. **Attack Models** - List the types of attacks considered within the design
6. **Related RFC Work** - Looks at similar RFCs and examines their application within WebSpa

This document concludes with a section summarising the findings of the sections stated above.

2 Use Cases

2.1 Objective

Similarly to traditional network port-knocking schemes, WebSpa aims to create a covert channel [**?**] of communication for O/S commands, over the web application layer. This channel is by no means bi-directional: It is only the client that can issue commands to the server. The inverse i.e. the server issuing commands to the client, is not an option within the current version.

What's more, the covert channel of communication established with WebSpa is very limited and can only be used for a set of premeditated actions. These actions must have been defined as O/S commands, within the server-side component of WebSpa, prior to a user attempting to issue a request. As a result, residues of steganography within WebSpa are kindly dismissed: Like TCP/IP, the protocol exhibits "sufficient structure and non-uniformity to be efficiently and reliably differentiated from unmodified ciphertext". [**?**]

Even though facilitating a very specific covert channel over the web application layer, WebSpa should not be utilized for steganographic purposes. The objective of the protocol is for the server to execute, under certain conditions, an O/S command. Note that direct O/S command execution is not an option. The conditions under which a command will execute are that it is already known and pre-defined, it is received by the server within a set time window of being issued by the client and that the corresponding user it authorized to execute it. We will examine each of the above controls in greater detail in the sections that follow.

The system definition for WebSpa is modeled on that of an existing website or web application, which any user can browse to, using their respective browser. [**?**]

2.2 Actors

Based on this, we can model WebSpa as having four (4) actors in total, three (3) of which are human, vs. one (1) which is an external system. This is gives us the total of users and respective external systems with which WebSpa interacts with.

1. **Website User** This actor has the role of a normal website user browsing the given website on which WebSpa is running on. They are expected to be using a web browser for issuing HTTP/S requests and receiving respective responses
2. **WebSpa User** More specific to the generalized actor 'Website User', this actor has the role of issuing specially crafted HTTP/S requests, which enable O/S system commands to be executed server-side.

3. **WebSpa Administrator** The actor responsible for setting up WebSpa users with respective pass-phrases and assigning O/S commands to action numbers. The administrator shares a secret pass-phrase with each user. This pass-phrase has been exchanged securely, using a different medium of communication with the user.

4. **Run O/S Command** The ability *"of the website"* to issue O/S commands to the host operating system, in the event of receiving a correctly crafted WebSpa request via HTTP/S.

Based on these actors, the following diagram depicts the use case for WebSpa, as a generalization of the use case of a website user browsing a particular website.

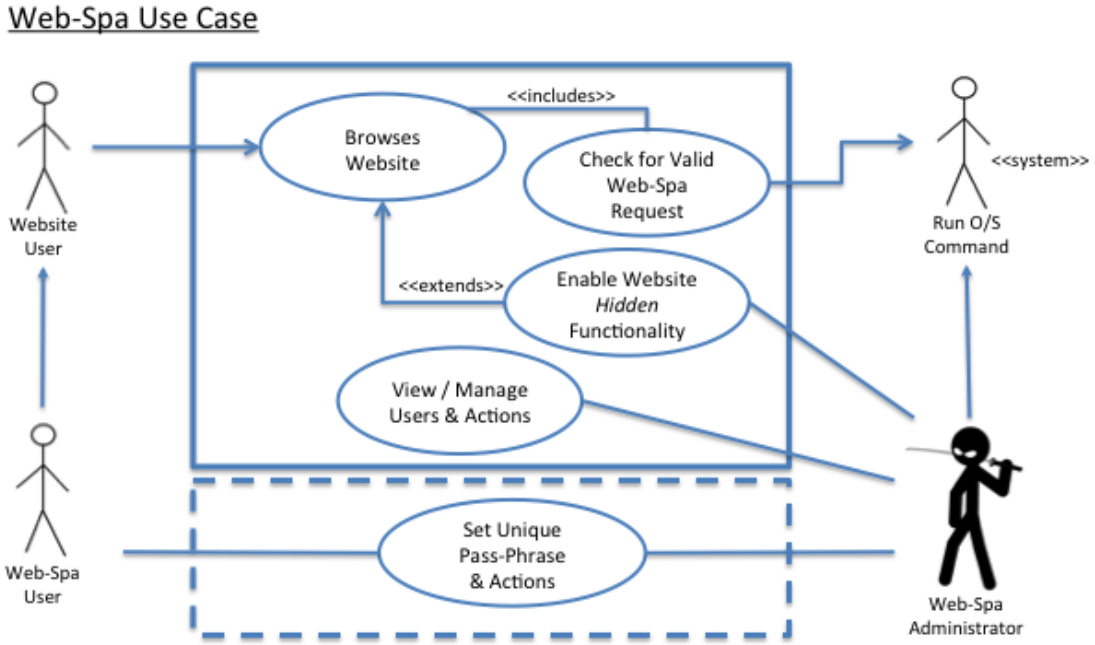


Figure 2: The WebSpa Use Case as an Extension of a Website Use Case

Overall, we can see that WebSpa is an encapsulating system that is expected to operate and function in the presence of a legimate publicly available website. Certain types of specially crafted requests issued to the site by a user of it will have the ability to execute respective O/S commands.

In order for that to take place, it is expected that an administrator of WebSpa is placed responsible for managing the respective users who have access to issue O/S commands.

3 Requirements Analysis

The procedure followed in the analysis of requirements for WebSpa is based on the 15 requirements analysis tasks listed in IEEE P1220 [**?**]. The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [**?**].

An implementation is not compliant if it fails to satisfy one or more of the MUST or REQUIRED level requirements for the protocols it implements. An implementation that satisfies all the MUST or REQUIRED level and all the SHOULD level requirements for its protocols is said to be "unconditionally compliant"; one that satisfies all the MUST level requirements but not all the SHOULD level requirements for its protocols is said to be "conditionally compliant."

3.1 Customer Expectations

The likely customer base of WebSpa is due to be paranoid individuals, with familiarity in the area of system administration. They will want something that works and does exactly what it says on the tin: nothing more; nothing less.

Using WebSpa is an action based on trust between the user and the administrator. As such, we begin by defining the requirements between these two type of customers in order for WebSpa to function correctly.

ID	Date	Version	Description
0101	13-Jan-2013	0.5	Each WebSpa **user** MUST be issued with a single secret pass-phrase, given to them once and in a typical out-of-band fashion, by the WebSpa **administrator**.
0102	13-Jan-2013	0.5	Each WebSpa **user** SHOULD be given a list of available actions by the WebSpa **administrator**.

Requirements 101 & 102 reflect also the design of WebSpa in being a 1 Factor Authentication (1FA) system. This implies that the user is only tied into the system by means of something they know and not by means of something they have or are. A number of discussions taken place with regards to introducing a second factor of authentication. These are intentionally omitted from this document.

Given that WebSpa requests are time dependant variables, the condition for knowing the current time in minutes exists.

ID	Date	Version	Description
0103	27-Apr-2013	0.7	Both the WebSpa **user** and **administrator** MUST know or be able to derive the current Unix time (i.e., the number of seconds elapsed since midnight Universal Time Coordinated (UTC) of January 1, 1970) in minutes.

Given that it is assumed any attacker will also be able to derive the current time in minutes, the confidentiality of any WebSpa request assumes the secrecy of the pass-phrase.

ID	Date	Version	Description
0104	28-Apr-2013	0.7	Both the WebSpa **user** and **administrator** MUST either share the same secret pass-phrase or the knowledge of a secret transformation to generate a shared secret pass-phrase.
0105	28-Apr-2013	0.7	Each WebSpa **user** MUST have a unique secret pass-phrase, known only to them and the WebSpa **administrator**.
0106	28-Apr-2013	0.7	The secret pass-phrase SHOULD be randomly generated or derived using a key derivation algorithms.
0107	28-Apr-2013	0.7	The secret pass-phrase MAY be stored in a tamper-resistant device and SHOULD be protected against unauthorized access and usage.

Finally, we stress the importance of the secret pass-phrase to carry enough entropy:

ID	Date	Version	Description
0108	28-Apr-2013	0.7	Each WebSpa **user** MUST use a strong secret pass-phrase. The length of the pass-phrase SHOULD be at least 128 bits. This document RECOMMENDs a pass-phrase length of 512 bits.
0109	06-May-2013	0.7	The WebSpa **user**'s secret pass-phrase MUST support all UTF-8 characters. This document RECOMMENDs a pass-phrase containing letters from foreign alphabets (e.g. the Greek letter α, or the letter Θ, etc).

This aims to aid customer expectation to not have simple pass-phrases such as ASCII based phrases like 'Password!' being used for web knocking operations. Using UTF-8 characters, even if it is for the purpose of making a phrase 'l33t', adds a lot more entropy to each character. Consider as an example 'PαsswΘrδ!' to 'P4ssw0rd!'.

3.2 Project and Enterprise Constraints

The constraints present impacting the design of the solutions follow the principle of least privilege.

ID	Date	Version	Description
0201	27-Apr-2013	0.7	The **server implementation** of WebSpa MUST NOT allow for the transmission of any data on the network.
0202	27-Apr-2013	0.7	The **client implementation** of WebSpa MUST NOT allow for the transmission of any data on the network, without the explicit permission from the user, granted for each transmission.

For every request received by the WebSpa server, an iterative hashing operation is executed on the pass-phrase of each WebSpa user. This hashing operation, despite not being computationally expensive (see section 5.6 entitled *Availability Attacks*) does mean that an upper limit needs to be set on the number of WebSpa users who can exist per WebSpa server instance.

ID	Date	Version	Description
0203	07-Sep-2013	0.7	The **server implementation** of WebSpa MUST NOT be allowed to run with more than 20 WebSpa users.

We recommend that no more than 20 WebSpa users are registered for a single server instance of this tool.

3.3 External Constraints

ID	Date	Version	Description
0301	13-Jan-2013	0.7	The WebSpa **protocol** MUST NOT use UDP.
0302	27-Apr-2013	0.7	The WebSpa **protocol** MUST be fully RFC 1945 (Hypertext Transfer Protocol – HTTP/1.0) compliant.
0303	09-May-2013	0.7	It is RECOMMENDED that the **implementation** of WebSpa is deployed over HTTPS (RFC 2818).

3.4 Operational Scenarios

There is absolutely no need for WebSpa to have any level of privileged access within the O/S kernel. Ultimately, WebSpa is a log monitor tool that issues commands to the host O/S. As such:

ID	Date	Version	Description
0401	13-Jan-2013	0.7	The **implementation** MUST NOT run in the O/S kernel.

The administrator of WebSpa is responsible for the commands which are made available to the users. This does not imply that by default these commands need to have *sudo* or other high levels of privilege.

3.5 Measure of Effectiveness (MoEs)

ID	Date	Version	Description
0501	13-Jan-2013	0.7	The **protocol** MUST establish a client-to-server channel of communication for issuing O/S commands over HTTP/S.

3.6 System boundaries

ID	Date	Version	Description
0601	13-Jan-2013	0.7	The WebSpa **protocol** SHOULD define a single HTTP request for one O/S command.
0602	05-May-2013	0.7	The WebSpa **server** MUST allow for O/S commands to be specified as actions, regardless of which underlying O/S it is running on.

3.7 Interfaces

In this subsection we describe the interfaces WebSpa depends on. Internal interfaces are those that address elements inside the boundaries established for the system addressed. These interfaces are generally identified and controlled by the contractor responsible for developing the system. External interfaces, on the other hand, are those which involve entity relationships outside the established boundaries. [**?**] [**?**]

3.7.1 External Interfaces the JAR requires to run

Operating System
WebSpa only has an indirect dependency on the operating system, as it written in Java. The operating system must be able to host and run the Java Runtime Environment (JRE).

ID	Date	Version	Description
0701	16-Jun-2013	0.7	The **implementation** MUST NOT have any direct dependencies within the underlying O/S.
0702	16-Jun-2013	0.7	The **operating system** MUST be able to run the Java Runtime Environment (JRE) in version 1.6 or greater.

Java Runtime Environment
In order to execute WebSpa's single Java archive file (.jar), the Java Runtime Environment must be installed on top of the server's operating system.

ID	Date	Version	Description
0703	16-Jun-2013	0.7	The **server implementation** MUST run on any JRE newer than version 1.6.
0704	16-Jun-2013	0.7	The **JRE** MUST be installed on the server.

Web Server
WebSpa does not accept any direct connections, but is rather designed as a log listener. When running in server mode, it needs to know and have access to the log file of the web server.

In server mode, WebSpa also needs to know what the regular expression specifying how to extract the originating IP Address of the request and the actual URL string is.

ID	Date	Version	Description
0705	16-Jun-2013	0.7	The **server implementation** SHOULD be able to parse log files from any web server.
0706	28-Aug-2013	0.7	The **server implementation** SHOULD allow for the location of the web server log file to be specified in a properties file named: **web-spa-config.properties**.
0707	28-Aug-2013	0.7	The **server implementation** SHOULD allow for the regular expression 'grepping' the IP Address and the URL to be specified in a properties file named: **web-spa-config.properties**.

3.7.2 Internal Interface the JAR depends on to run

Config File

ID	Date	Version	Description
0708	16-Jun-2013	0.7	The config file of the **server implementation** MUST be named **web-spa-config.properties** and be at the time of launch in the same directory as the jar file of Web-Spa.
0709	28-Aug-2013	0.7	A default config file for the **server implementation** MUST be created during launch if the file **WebSpa-config.properties** does not exist in the same directory relative to the location where the .jar file resides.

Database Files

In server mode, WebSpa also carries an embedded database, used to store webknocks received, user pass-phrases and respective actions.

ID	Date	Version	Description
0710	16-Jun-2013	0.7	The database file of the **server implementation** MUST be named with the prefix **web-spa-db** and be at the time of launch in the same directory as the jar file of WebSpa.
0711	28-Aug-2013	0.7	The two default database files **web-spa-db.properties** & **web-spa-db.script** MUST be created during launch if these files do not exist in the same directory relative to the location where the .jar file resides.

Time
A time value is used as a salt for the *digest* function. Therefore, it is essential that both, the WebSpa client and the WebSpa server share the same time.

ID	Date	Version	Description
0712	16-Jun-2013	0.7	Both, the client and the server **implementations** MUST share the same system time.
0713	16-Jun-2013	0.7	The client and the server **implementations** SHOULD use UTC time in order to prevent wrong calculations caused by differing time zones.

3.8 Utilization environments

ID	Date	Version	Description
0801	13-Jan-2013	0.7	The WebSpa **protocol** MUST use known cryptographic techniques and be based around the concepts of a hash commit.

3.9 Life cycle

Despite being described often as a mythical goal, we are going to set the bar at a very high level when it comes to test cases and code coverage.

ID	Date	Version	Description
0901	13-Jan-2013	0.7	The crypto library **code** MUST have 100 percent test code coverage with respective test cases for its functionality.

Future versions of WebSpa should also have a much higher test code coverage, not focusing solely on the crypto library, but the wider code base of the executable.

Type safety is also very important in the selection of the programming language used to code the server implementation of WebSpa.

ID	Date	Version	Description
0902	06-May-2013	0.7	The **server implementation** MUST NOT be written in an unsafe language.

3.10 Functional Requirements

3.10.1 Protocol

ID	Date	Version	Description
1001	13-Jan-2013	0.7	The WebSpa **protocol** MUST establish a client-to-server channel of communication for issuing O/S commands over HTTP/S.
1002	13-Jan-2013	0.7	The WebSpa **protocol** MUST NOT allow for direct O/S command execution on the server.
1003	13-Jan-2013	0.7	The WebSpa **protocol** MUST NOT allow for the server to issue O/S commands to the client.
1004	13-Jan-2013	0.7	Each O/S **command** MUST be already known by the WebSpa server before being issued by the client.
1005	31-Jan-2013	0.7	Each WebSpa **request** MUST result in the execution of an O/S command only once.

For the benefit of simplicity of the WebSpa protocol and to save system resources when evaluating incoming requests, it was decided to identify the WebSpa user based on the pre-shared secret.

ID	Date	Version	Description
1006	21-Jul-2013	0.7	Each **shared secret** MUST be unique in order to distinguish the users.

3.10.2 Server

ID	Date	Version	Description
1007	13-Jan-2013	0.7	The **server implementation** MUST keep a clear record of O/S commands executed.
1008	13-Jan-2013	0.7	The **server implementation** MUST have a single configuration file, located in **web-spa-config.properties** relative to the path of the stand-alone executable.
1009	13-Jan-2013	0.7	The **server implementation** data MUST have a single database file, located in **web-spa-db.data** relative to the path of the stand-alone executable.
1010	13-Jan-2013	0.7	The **server implementation** MUST NOT use libpcap and MUST NOT inspect every packet.
1011	31-Jan-2013	0.7	The **server implementation** MUST NOT have a new service and MUST NOT bind to any port.

3.11 Performance requirements

ID	Date	Version	Description
1101	01-May-2013	0.7	The **server implementation** of WebSpa MUST have run without human intervention for 100 hours, averaging 10 valid requests per hour.
1102	01-May-2013	0.7	The **server implementation** of WebSpa MUST have run for without human intervention for 1000 hours, without receiving a valid request for at least 3 time intervals of 168 hours.
1103	01-May-2013	0.7	Both the **server implementation** and the **client implementation** SHOULD have a very small memory and execution footprint. The performance requirements SHOULD be minimal. .

3.12 Modes of operation

As such we have to provide something simple:

ID	Date	Version	Description
1201	27-Apr-2013	0.7	The **server implementation** of WebSpa SHOULD be a single executable file.
1202	27-Apr-2013	0.7	The **implementation** of WebSpa MUST include a client component, responsible for generating WebSpa requests.
1203	27-Apr-2013	0.7	The WebSpa **server implementation** SHOULD include a component, responsible for monitoring web server logs for WebSpa requests.

3.13 Technical performance measures

ID	Date	Version	Description
1301	16-Jun-2013	0.7	The **server implementation** of WebSpa MUST be executable on a mid-range server.

3.14 Physical characteristics

Given the above described modes of operation, we can follow very strict guidelines as to the physical characteristics of the single executable for both the WebSpa client and server.

ID	Date	Version	Description
1401	08-Jul-2013	0.7	All functionality of the WebSpa executable SHOULD be contained within the one file.

The above implies that shipping a single jar file as one download, should suffice. What is more, filesize should be taken into account, as we don't want to over-engineer the specification. Ergo:

ID	Date	Version	Description
1402	08-Jul-2013	0.7	The filesize for the single executable file of WebSpa SHOULD not exceed a total of 5Mb.

Finally, control should be given to the user to modify and adjust WebSpa to their liking and needs. The tinkering of WebSpa is most welcome; on the client, there is not too much to tinker, while on the server there are a few things one can manipulate.

ID	Date	Version	Description
1403	08-Jul-2013	0.7	The **client implementation** of WebSpa SHOULD NOT require any additional files present to run.

While on the server matters are a little bit different. There exists a corresponding configuration file, as well as an embedded database element.

ID	Date	Version	Description
1403	08-Jul-2013	0.7	The **server implementation** of WebSpa SHOULD create within the local folder of execution all necessary files for WebSpa to operate correctly.

As a user of WebSpa you are expected to download a zip file. Documentation aside, this zip is likely to contain a jar file inside a WebSpa folder. Upon running the client, no further files, folder or temp files get created. Upon running the server, all required files get placed in the current directory of execution.

3.15 Human systems integration

ID	Date	Version	Description
1501	16-Jun-2013	0.7	Both, the **server implementation** and the **client implementation** MUST provide either a command line interface (CLI), or a graphical user interface (GUI) to interact with the user.

4 The Request

In the simplest scenario, a user is attempting to run a pre-defined O/S command on a standard web server. We refer to this command as an action. For this, the client will generate a URL from a number of inputs provided by the user; we refer to this as the WebSpa request.

4.1 Request Structure

The WebSpa request uses a secret **pass-phrase** and a pre-defined **action** number to generate a sequence of one-time (single use) URL characters. One more additional input is used by the system performing the generation. This is the number of minutes in epoch (UTC) time.

In summary, three **(3)** inputs are required in any one time to generate a WebSpa request.

- [user provided] pass-phrase
- [user provided] action
- [system] UTC time (minutes)

These inputs result in an HTTP Base64 URL-Safe encoded URL being generated. Each request within the URL has the same identical length of **100** characters, carrying a total of **75** bytes of data. Each request changes every minute, given the change of the minutes. The structure of a WebSpa request is as follows:

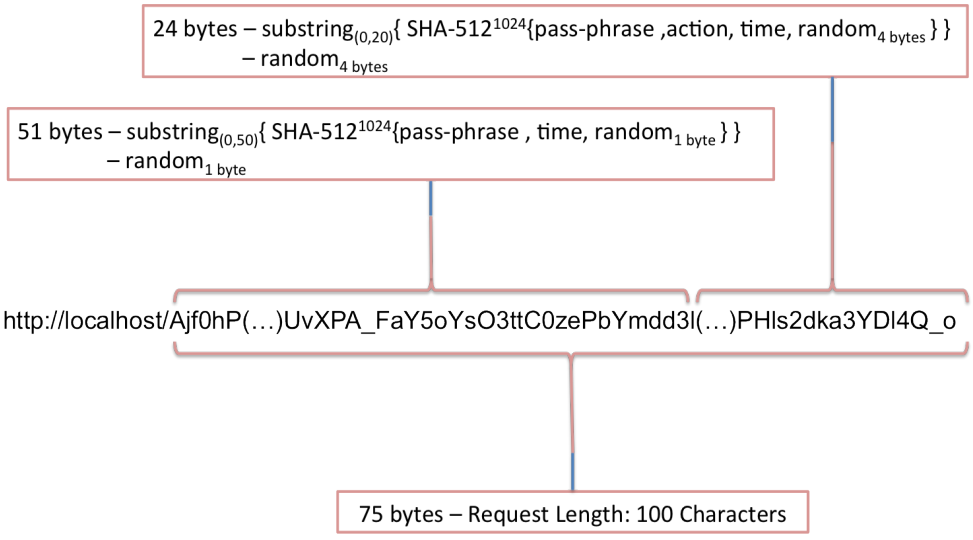


Figure 3: WebSpa Request Structure

A WebSpa request consists of two parts. We refer to each of the different parts of the request as the **elements** of the WebSpa request and proceed to number them accordingly. By concatenating the bytes of the two elements together, we obtain a total of 75 bytes of data. This yields a total of 100 characters of Base64 URL safe encoded bytes for each request. [**?**] [**?**]

In the following sections, we iterate through how each element is derived. Following that, the design rationale and data processing steps are presented.

4.2 Element 1: Pass-knock

The pass-knock element (not be confused with the user's pass-phrase) is the result from the concatenation of one random byte with a truncated SHA-512 hash. The truncated SHA-512 hash is reduced to 50 bytes in length. Thus, a pass-knock element has a total length of 51 bytes: 50 from the truncated SHA-512 hash and 1 random byte.

The steps followed to derive the pass-knock are:

1. We use the user's pass-phrase and the UTC time (minutes)
2. We obtain the UTF-8 bytes from the pass-phrase
3. We obtain the 4 bytes of the current minute
4. We place all the bytes in an array and sort it
5. We generate 1 random byte value
6. We concatenate the sorted array with the random byte
7. We SHA512 the above array a total of 1024 times
8. We obtain the first 50 bytes from the above
9. We concatenate the one random byte with the first 50 bytes

In the above steps the UTF-8, sort, concatenate and SHA512 byte functions represent standard functions. It is worth noting that the order in which the concatenation of any two byte arrays is important. Obtaining the current minute in a byte array of 4 elements requires further explanation:

1. We use the current UTC time in minutes
2. We obtain the current number of milliseconds
3. We divide by (60 * 1000)
4. We convert the result into a byte array
5. From this array we obtain the last 4 entries

These 4 bytes are concatenated to the UTF-8 bytes of the user's secret pass-phrase. The value of the concatenated bytes is then sorted.

An example implementation in Java is presented below:

```
1  public static byte[] getHashedPassPhraseInTime(CharSequence passPhrase,
       long currentTimeMinutes) {
2
3      byte[] passBytes = passPhrase.toString().getBytes(Charsets.UTF_8);
4      byte[] timeBytes = ByteBuffer.allocate(8).putLong(currentTimeMinutes).
           array();
5
6      byte[] sortedBytes = new byte[passBytes.length + timeBytes.length - 4];
7      System.arraycopy(passBytes, 0, sortedBytes, 0, passBytes.length);
8      System.arraycopy(timeBytes, 4, sortedBytes, passBytes.length, timeBytes.
           length - 4);
9
10     Arrays.sort(sortedBytes);
11
12     SecureRandom scRandom = new SecureRandom();
13     byte[] randomByte = new byte[1];
14     scRandom.nextBytes(randomByte);
15
16     byte[] allBytes = EncodingUtils.concatenate(sortedBytes, randomByte);
17     byte[] hashedBytes = ArrayUtils.subarray(digest(allBytes), 0, 50);
18
19     return EncodingUtils.concatenate(randomByte, hashedBytes);
20
21 }
```

With the method digest() performing a total of 1024 iterations of the SHA-512 algorithm:

```
1  public static byte[] digest(byte[] value) {
2
3      for (int i = 0; i < 1024; i++) {
4         value = DigestUtils.sha512(value);
5       }
6
7      return value;
8  }
```

The above implementation uses the class DigestUtils from the Commons Codec (_v1.8 API). Any standard implementation for the SHA-512 algorithm can be used as well.

4.3 Element 2: Action-knock

The action-knock element (not to be confused with the user's action) is the result from the concatenation of four random bytes with a truncated SHA-512 hash. The truncated SHA-512 hash is reduced to 20 bytes in length. Thus, an action-knock element has a total length of 24 bytes: 20 from the truncated SHA-512 hash and 4 random bytes.

The steps followed to derive the action-knock are:

1. We use the user's pass-phrase, action and the UTC time (minutes)
2. We obtain the UTF-8 bytes from the pass-phrase
3. We obtain the 1 byte of the user's action
4. We obtain the 4 bytes of the current minute
5. We place all the bytes in an array and sort it
6. We generate 4 random byte values
7. We concatenate the sorted array with the 4 random bytes
8. We SHA-512 the above array a total of 1024 times
9. We obtain the first 20 bytes from the above
10. We concatenate the 4 random bytes with the first 20 bytes

In the above steps the UTF-8, sort, concatenate and SHA-512 byte functions represent standard functions. Once again, it is worth noting that the order in which the concatenation of any two byte arrays is important.

The way in which a byte array of 4 bytes is derived from the current minute has been described in the previous section. The digest function has been also described in the previous section.

Obtaining the user's action as a single byte requires further explanation:

1. We use the user's action (represented as an Integer)
2. We convert the user's action into a byte array of 4 entries
3. From this array we obtain the last byte entry

This implementation implies that only the numbers [0, 127] inclusive will be converted to valid byte numbers of equal value. This is because in Java:

```
1 byte[] actionBytes = ByteBuffer.allocate(4).putInt(127).array();
2 [0, 0, 0, 127]
```

While:

```
byte[] actionBytes = ByteBuffer.allocate(4).putInt(128).array();
[0, 0, 0, -128]
```

The single action byte is concatenated to the UTF-8 bytes of the user's secret pass-phrase and the 4 time bytes. The value of the concatenated bytes is then sorted.

An example implementation in Java is presented below:

```
public static byte[] getHashedActionNumberInTime(CharSequence passPhrase,
    int actionNumber, long currentTimeMinutes) {

    byte[] passBytes = passPhrase.toString().getBytes(Charsets.UTF_8);
    byte[] actionBytes = ByteBuffer.allocate(4).putInt(actionNumber).array()
        ;
    byte[] timeBytes = ByteBuffer.allocate(8).putLong(currentTimeMinutes).
        array();

    byte[] sortedBytes = new byte[passBytes.length + timeBytes.length - 4 +
        1];
    System.arraycopy(passBytes, 0, sortedBytes, 0, passBytes.length);
    System.arraycopy(timeBytes, 4, sortedBytes, passBytes.length, timeBytes.
        length - 4);
    System.arraycopy(actionBytes, actionBytes.length - 1, sortedBytes,
        sortedBytes.length - 1, 1);

    Arrays.sort(sortedBytes);

    SecureRandom scRandom = new SecureRandom();
    byte[] randomByte = new byte[4];
    scRandom.nextBytes(randomByte);

    byte[] allBytes = EncodingUtils.concatenate(sortedBytes, randomByte);
    byte[] hashedBytes = ArrayUtils.subarray(digest(allBytes), 0, 20);

    return EncodingUtils.concatenate(randomByte, hashedBytes);

}
```

4.4 Design Rationale

At the heart of WebSpa lies a Keep It Simple, Stupid (KiSS) cryptographic commitment scheme [**?**] that is used to generate the pass-knock and the action-knock.

WebSpa avoids the use of symmetric keys and also stays well away from any asymetric cryptosystems to use the kitchen sink of modern cryptography for a single authorisation request. There is simply no need for an 'open-sesame' type of protocol that it is.

What is more, we have eliminated the need for a user name. If the user knows who they are through a unique pass-phrase to the system, there is no need for a second input to be provided by them. As the issuing of pass-phrases is regulated by an administrator and given that the administrator is the regulating body for which O/S commands are available to the user, we didn't see the need for the user name field to exist.

In taking the current minute in UTC as input from the system, we ensure that the value of the pass-phrase is not used without a time-dependant variable, thus varying every 60 seconds.

Like in RFC 6238, entitled TOTP: Time-Based One-Time Password Algorithm, the WebSpa algorithm for each element is based on the SHA hash algorithm and truncated. WebSpa uses a stronger variant (i.e. SHA-512) over a total of 1024 iterations.

For each element of WebSpa, a randomly selected input is padded onto the value that will be SHA-512 hashed a total of 1024 iterations. This randomly selected input is then transmitted in the clear as part of the WebSpa request and serves as a salt.

To sum up to a total of 75 bytes (100 base64 characters), the first element of WebSpa (pass-knock) is 51 bytes and the second element (action-knock) is 24 bytes.

A variant of base64 that is URL safe is used for the encoding and decoding of WebSpa messages. This does not chunk the output. Also, this substitutes the characters + and / with - and _ respectively.

Further analysis on each of the components mentioned above can be found in the section entitled RFC & Other Related Work.

4.5 Data Processing Steps

The functional block diagram shown in the figure below illustrates the dataflow from client to server for WebSpa. This diagram is based on the 1983 IEEE block diagram of a typical digital communication system.

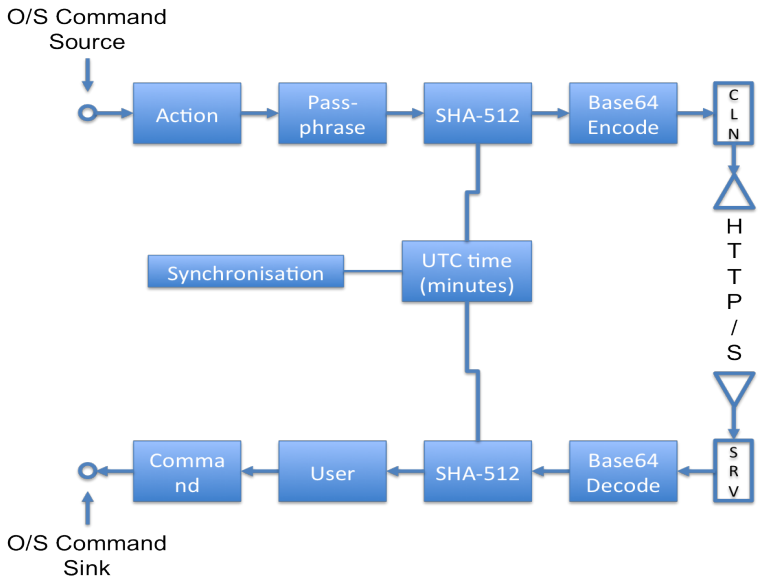


Figure 4: The functional block diagram of WebSpa

The upper blocks, labelled 'Action', 'Pass-phrase', 'SHA-512' and 'Base64 Encode' dictate data transformations from the client to the transmitter. The lower blocks dictate the signal and data transformations from the receiver back to the source, essentially reversing the processing steps performed by the upper blocks.

The block described as 'SHA-512' performs a slightly more complex operation than generating the output of the SHA-512 algorithm. This operation is described in detail for each element in the sections above.

Finally, the dashed block around the blocks of 'Action' and 'Pass-phrase' indicates the user's privacy: These two inputs are provided by the user for each WebSpa request to the client and are not transmitted in the clear.

5 Attack Models

No system is totally secure; WebSpa is no exception to this statement. Still, we have tried to make it very hard for a determined attacker to compromise a server running WebSpa.

In this section we list the hypothesis, attack trees and conditions under which WebSpa can operate in a secure way. We also list a number of valid attacks that can be performed on a WebSpa server that would render our security controls useless.

After having had endless discussions, re-designed the crypto libraries several times and lost some of the precious hair left on our foreheads, we believe that WebSpa is now prone to withstand a number of attack scenarios. Here is the How & Why.

5.1 Hypothesis

We start off by defining what we take for granted. Compromise any of the components listed in the next paragraph and WebSpa would probably be left dead in the water.

Our hypothesis is that the guest O/S, respective web server and underlying java runtime have been configured and are running in a secure way. You probably should not be attempting to set up a covert channel of communication, if the primary channel is not one that you trust and believe to be secure.

Yes, there will always be the occasional zero-day vulnerability and respective plethora of exploits, but what this hypothesis is stating is that if you have enabled a telnet port with default username/password, you probably have bigger problems than safeguarding the security your WebSpa server.

This hypothesis implies that each of those components are *pass-through* components of tainted data; WebSpa should be able to deal with such data flows.

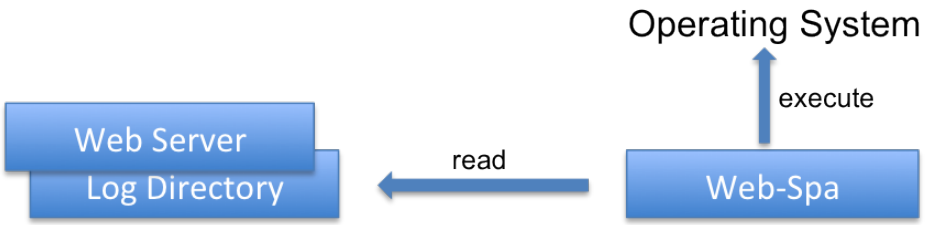


Figure 5: The system architecture of WebSpa

WebSpa does not listen on any port and does not accept direct connections. It is designed as a log listener whereby it 'tails' the log file of a web server for new entries in a certain format. The above should help set the respective one-way-trust relationships between each of the components mentioned.

5.2 Attack Trees

An attack tree is defined as a methodical technique of enumerating the *what if* scenarios based on varying attacks under a common goal: To breach a system.

Since WebSpa is a complex program, it has a fairly complex tree under the primary goal of *"execute an O/S Command on a server running WebSpa"*. We have build this attack tree taking into account the 1999 Schneier PGP example attack tree found in [**?**].

Goal 1: Execute an O/S Command on a server running WebSpa

1. Decrypt the web-knock message itself (OR)
   - 1.1. Break hash commit operation (OR)
     - 1.1.1. Brute-force hash commit operation (OR)
     - 1.1.2. Mathematically break hash commit operation (OR)
       - 1.1.2.1. Break SHA-512 (OR)
       - 1.1.2.2. Break partial SHA-512 substring used (OR)
2. Determine pass-phrase or action used to generate the web-knock via other means (OR)
   - 2.1. Fool WebSpa user into sending their pass-phrase into the clear (OR)
     - 2.1.1. Convince the WebSpa user to use a fake WebSpa client (OR)
   - 2.2. Monitor the memory of the computer where the WebSpa client is used (OR)
   - 2.3. Monitor the memory of the computer where the WebSpa server is used (OR)
   - 2.4. Implant virus that exposes the pass-phrase and action number (OR)
3. Get the WebSpa administrator to (help) decrypt the message (OR)
   - 3.1. Chosen ciphertext attack on the web-knock (OR)
   - 3.2. Send another user's pass-phrase and action number (OR)
   - 3.3. Read WebSpa server database
     - 3.3.1. Copy database off the administrator's hard drive or virtual memory (OR)
     - 3.3.2. Copy database from back locations (OR)
     - 3.3.3. Monitor administator's network traffic (OR)
     - 3.3.4. Use electromagnetic/wireless snooping techniques to read messages as it is displayed on the screen (OR)
   - 3.4. Compromise the 'out-of-band' channel of communication used between the WebSpa user and administrator

In failing to execute an O/S command on a server that is running WebSpa, the second objective would be to make that server unresponsive to all other WebSpa requests, thus disabling the covert channel of communication.

Goal 2: Make a server running WebSpa unresponsive to all requests

1. Crash the server so that to force access via an insecure channel (OR)
   1.1. Send so many requests that the web server becomes unresponsive (OR)
   1.2. Send so many WebSpa requests that the WebSpa server becomes unresponsive (OR)
      1.2.1. Create a valid web-knock from an active user's pass-phrase and action number that is submitted repeatedly to the WebSpa server (OR)
      1.2.2. Create invalid web-knock requests that pass the basic input validation tests and submit them repeatedly to the WebSpa server (OR)

In the event of not knowing which server to attack, a determined attacker would have to be able to identify a WebSpa server first.

Goal 3: Identify a server that is running WebSpa

1. Periodically monitor a server for changes in its services (OR)
2. Compromise a server and find the respective WebSpa server executable, database and configuration files on it

In the sections that follow, we focus on what we believe to be the most successful of these attacks, in terms of achieving one of the above 3 stated goals. For each of these attacks, we list the controls, defences and conditions required for each attack to be successful.

5.3 Timed Brute Force Attacks

Like any single password based authentication system, WebSpa is vulnerable to a brute force attack. A malicious attacker iterates through a dictionary of potential pass-phrases, encodes them (using the WebSpa client) as valid web-knocks and attempts to identify if one of them is a valid user's pass-phrase.

There are two defences to this attack build in the design of the WebSpa. First, the malicious attacker would have to know which action number to select for a particular O/S command and second, the malicious attacker would have to know what that O/S command actually was.

The former of these two defence is not really a defence; needing to select an action number in the range of [0 - 9] merely adds another factor to the brute-force attack. The reason why we consider it as a defence is because unlike a username/password screen

where once the right combination is submitted, you get granted access, with WebSpa there are 10 possible potential states that you have to monitor at any one time in order to see if the combination of pass-phrase and action number is one that is valid.

The latter of these two defences actually implies that in order to successfully execute a timing brute force attack against a server running WebSpa, you would have to know of at least one O/S command that when executed would change the state of the server in such a way that it would be noticeable from an external attacker's perspective.

Both these two defences can be bypassed in the event of a malicious attacker coming in contact with or seeing how WebSpa is used on real 'live' server. They can also be bypassed if a malicious attacker colludes with the WebSpa administrator.

A timed brute force attack on a covert channel of communication is something that WebSpa is vulnerable to. This is why it is paramount that strong pass-phrases are selected and given to each WebSpa user.

5.4 Replay Attacks

A replay attack on a machine running WebSpa in *server mode* is an attack that allows for a passive or active eavesdropper to capture, delay and replay web-knocks being sent from the client to the server.

If we ignore the recommendation of setting up WebSpa to monitor the log file of a web-server that is configured to work solely over the HTTPS protocol, one additional defence mechanism has been build into the tool.

As specified in section 4.4 entitled *Design Rationale*, a token is valid for 60 seconds. Therefore, even in the event of not using WebSpa over HTTPS, a malicious attacker intercepting a valid web-knock, will only have 60 seconds to re-use this token, before it becomes redundant.

5.5 Man-in-the-middle Attacks

A man-in-the-middle attack on a machine running WebSpa in *server mode* is an attack that allows for an active eavesdropper to impersonate one of the two endpoints of communication.

Continuing to ignore the recommendation of setting up WebSpa on HTTPS, there is one fundamental control preventing the replay attack scenarios discussed previously to expand into wider man-in-the-middle attacks.

There is a common secret that has been shared in the form of a pass-phrase between each WebSpa user and the WebSpa administrator.

Thus, in order for this attack to materialise in the form of a WebSpa server receiving a message that impersonates a user of WebSpa, knowledge of that user's pass-phrase must have been made available to the party impersonating them.

Inversely, for a WebSpa user believing that they are communicating with not the real WebSpa server by instead an imposing end-point, the server would have to know how to react to a web-knock, by knowing who the user is.

These conditions are not impossible to meet and can occur just by active observation under very set conditions. Examples would include a user always submitting a web-knock request from the same IP address and have a single action assigned to them.

As such, some form of collusion or otherwise accomplished compromise must be already happening for a man-in-the-middle attack to be able to succeed in a remit wider to that of a replay attack.

5.6 Availability Attacks

An availability (or resource starvation) attack on a machine running WebSpa in *server mode* is an attack that renders the WebSpa executable unresponsive based on one or more valid or invalid requests being received.

The controls preventing this attack from materialising are:

1. The line of text within the log file cannot be more than $2^{16} - 1$ characters

```
// Check if the line length is more than 65535 chars
if (requestLine.length() > Character.MAX_VALUE) {
  return;
}
```

2. There is a 10 second delay between checks of the log file for new content received

```
if(myLogTailer == null) {
  myLogTailer = Tailer.create(accessLog, myLogListener, 10000, true);
} else {
```

In addition to this, WebSpa can only process requests as fast as your web server of choice can write them to the log file. Thus, there is a definite protocol change between a user submiting a valid or an invalid WebSpa request being attempted to be processed by WebSpa.

5.6.1 User Iteration Availability Attack

One avenue of attack that also has to be explored relates to the fact that for a URL request that appears to be a valid WebSpa request (one that is 100 characters), the passphrase of *every* user will be fetched from the database. Once that fetching database event takes place, the hash value for the current minute will be calculated and compared to web-knock received.

This can lead to an attack caused by the fact that the WebSpa server becomes unresponsive while attempting to calculate every valid hash for that minute in time.

For the two steps required to execute this operation, our analysis (both in the form of JUnit testing as well as profiling the software) shows that fetching all the user's passphrases is a simple *SELECT* operation requiring less than 100 ms to complete.

Similarly, the JUnit tests for the class *net.selecus.wsp.crypto.WebSpaEncoder* clearly illustrate that the method (named matches) calculating a hash value requires approximately 04-10 ms to complete. Such short computational times are expected, given that no heavy duty crypto operations take place when calculating a web-knock, plain SHA-512 in an iterative way is the most expensive operation that actually takes place.

The bullets below, list the amount of time (in seconds) it takes to execute a matching exercise between a pass-phrase and a valid web-knock. These values are extracting from the respective JUnit tests.

- testMatchesFalse(net.seleucus.wsp.crypto.WebSpaEncoderTest) - (0.010 s)
- testMatchesTrue(net.seleucus.wsp.crypto.WebSpaEncoderTest) - (0.004 s)
- testEncode(net.seleucus.wsp.crypto.WebSpaEncoderTest) - (0.004 s)

Thus, even in the event of a WebSpa server having 100 active users, the operation of calculating their respective hash values for that particular minute in time is not one that is likely to take more than 1 second.

Based on these facts we do recommend that no more than 20 WebSpa users are registered for a single server instance of this tool.

To summarise, we do not believe that a user iteration attack can impact the availability of this tool, provided no more than the recommended number of users are registered with the WebSpa server.

Of course, WebSpa might be indirectly affected by Distributed Denial of Service (DDoS) & Denial of Service (DoS) attacks against the web server, or the underlying operating system.

5.7 Administrator Collusion Attacks

If the administrator of WebSpa decides to collude against a user of WebSpa with a malicious attacker, there are no protections in the design or implementation of this system that can assist the user.

A WebSpa adminstrator has access to the pass-phrase of each WebSpa user, their respective action numbers, assigned to respective O/S commands and also all the records of when each user executed each command, if it was successful, etc.

5.8 Vertical Privillege Escalation Attacks

A vertical privillege escalation attack on a machine running WebSpa in *server mode* is an attack that allows a legitimate WebSpa user to become a WebSpa administrator.

Such an attack can be quite common, if the legitimate WebSpa has been granted account access to the machine running WebSpa in *server* mode and that account has user file permissions which are the same or superset those that the WebSpa jar is running under.

Such an attack can be easily prevented by setting or assigning a user account to run the WebSpa jar, which does not allow other users to access this file location.

As WebSpa has been designed to create all database and configuration files within the directory that it is been executed, restricting file access to this location would immediately remediate against this attack.

5.9 Cryptanalytic Attacks

As a protocol of communication, WebSpa does allow for a form of known-plaintext attacks to materialise. An attacker intercepting a valid web-knock at a particular moment in time would have access to the following elements of plain-text information.

- The random byte(s) used
- The current UTC used

This gives an attacker two out of the three elements required, leaving the most important pass-phrase as the unknown element of the web-knock request. Variants of differential cryptanalysis techniques can be used, provided enough ciphertext is collected.

We have attempted to make a cryptanalyst's life a little bit more difficult by hashing our inputs a number of times. Both, the pass-knock and the user-knock are processed by the *digest()* function and doing so hashed with SHA512 for a total of 1024 times. As an additional security measure, random bytes are added prior to the hash-cycles to further increase entropy.

This design also prevents attacks based on precomputed tables, known as rainbow tables. Despite often been used for reversing cryptographic hash functions, rainbow tables generated for the standard hash function SHA-512 would not be much assistance in the cryptanalysis of WebSpa.

In other words, we believe that with the current level of technology, the *digest()* function can resist chosen plaintext attacks for a usable amount of time.

5.10 Rubber-hose Cryptanalysis

When establishing a covert channel of communication, based on a single *'something you know'* factor of authentication, that is shared as a common secret between the administrator and the user, rubber-hose cryptanalysis has to be mentioned as a potential attack avenue.

Given what WebSpa represents, it might unfortunately be easier to use "less sophisticated" methods of cryptanalysis such as simply beating the secret out of the key holder(s), or putting the most common forms of human corruption to good use.

6 RFC & Other Related Work

In this section, we note a number of building blocks on which the design of WebSpa has been based upon.

Firstly, for the concept of executing an O/S command via means of a single request, we reference fwknop [**?**]. Even though fwknop is a network access control (via iptables) tool, it implements an authorization scheme called Single Packet Authorization (SPA). The design of WebSpa is based on the concept of a single HTTP/S request, building on from the concept of SPA.

Secondly, as described in the design rationale section of the request, at the heart of WebSpa lies a KiSS cryptographic commitment scheme [**?**] that is used to generate the pass-knock and the action-knock. The basis of this hash operation stems from the Spring StandardPasswordEncoder Class [**?**]. In WebSpa the two knock sequences are generated via means of the hash techniques described in the Spring Class.

Thirdly, like RFC 6238, entitled TOTP: Time-Based One-Time Password Algorithm, there is a time changing element to each WebSpa request. Ergo, The concept of changing the request hash every 60 seconds stems from the TOTP RFC.

We examine these in greater detail in the sections below.

6.1 Fwknop and the Power of SPA

The idea of WebSpa came from reading the fwknop specification [**?**]. If a single TCP packet can be deemed sufficient to enable network level access of sorts, a single URL request over HTTP/S would surely carry the same potential but with less hassle.

Afterall, more people have access to a web browser and HTTP, in its early version (0.9 & 1.0) was a protocol with a specification of a single request being followed by a single response.

6.2 Spring StandardPasswordEncoder Class

While designing this version WebSpa and having learned from the mistakes of the previous version, we really wanted to keep matters simple. For this reason, we looked at the most standard way of deriving a password hash and what better place to start than the Java world of Spring Source [**?**].

Thus the concept of using an iterative hashing operation based on SHA-512 and seeding it with a random salt got introduced to WebSpa. This, in combination with the time based nature of requests described in the section below, provided a pillar of defences against a number of attacks.

6.3 RFC 6238

Despite not being a direct derivation of an existing RFC, WebSpa shares a number of common design characteristics mainly with RFC 6238, entitled TOTP: Time-Based One-Time Password Algorithm.

Like RFC 6238, WebSpa uses both a truncation function, as well as a well established hash algorithm, namely SHA-512. Unlike RFC 6238, which receives one user input, WebSpa takes as input two values: The user's secret pass-phrase and current action.

As WebSpa operates on UTC time measured in minutes, it could be argued that WebSpa is an RFC 6238 variant where T0=0 and X=60. One more reason amplifying that argument is that RFC 6238 allows for TOTP implementations that 'may' use HMAC-SHA-512 functions. Having said that WebSpa does not utilise HOTP (RFC 4226) as a building block.

Finally, unlike RFC 6238, WebSpa uses an iterative hashing operation (running the SHA-512 algorigthm iteratively, 1024 times) and seeds each hash with a random salt. The salt seeding the web-knock is one (1) byte in length, while the salt seeding the action-knock is four (4) bytes in length. Both salt values are transmitted with each WebSpa request.

7 Conclusions

WebSpa is designed as a log listener and does not accept any direct connections or implement any additional services. It scans the access log file of a web server for a specific pattern. This pattern consists of a unique pass-phrase for each user and a pre-meditated action representing an O/S command. We refer to this pattern as the web-knock.

A web-knock is a string of 100 Base64 URL-safe encoded characters. The first 68 characters represent a TOTP representation of the user's pass-phrase truncated to that length and changing every 60 seconds, while the last 32 characters represent a TOTP representation of the user's pre-meditated action and pass-phrase also changing every 60 seconds.

As a cryptosystem, WebSpa solely relies on the concept of a hash-commit, having as a base algorithm SHA-512. Each part of a web-knock is randomly salted and the SHA-512 hashing function is repeatedly applied. In order for this communication to work between the client and the server, both sides of the communication channel are expected to be in sync with the current UTC time.

Each 100 Base64 URL-safe encoded characters, is checked in two ways: First, in terms of authentication i.e. has a valid pass-phrase being received at that moment in time and second, in terms of authorisation i.e. has the user with the respective pass-phrase have the corresponding right to execute the particular action. If so, the requested O/S command is executed and logged on the WebSpa server.

The executable itself is a single lightweight .jar file that is currently less than 4MB in size and uses only a negligible amount of system resources. During stress testing, even processing 1000 WebSpa requests at a time - which is very unlikely to happen out in the wild - did not affect the system performance of a mid-range server.

The creators of WebSpa have spent countless hours discussing various approaches and design concepts. We believe in having created a secure way to execute critical O/S commands on a remote web server through a covert channel, without the need for exposing additional open ports or requiring to run more services.

Thank you very much for your interest in WebSpa.

8 References

9 Abbreviations

1FA	1 Factor Authentication
AES	Advanced Encryption Standard
CLI	Command Line Interface
DDoS	Distributed Denial of Service
DoS	Denial of Service
GUI	Graphical User Interface
HTTP/S	Hypertext Transfer Protocol / Secure
KiSS	Keep It Simple, Stupid
HMAC	(Keyed-) Hash Message Authentication Code
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
IP	Internet Protocol
JAR	Java ARchive
JRE	Java Runtime Environment
NCSC	National Cyber Security Centrum
NSA	National Security Agency
O/S	Operating System
OTP	One-Time Password
OTT	One-Time Token
RFC	Request for Comment
SHA	Secure Hash Algorithm
SPA	Single Packet Authorization
TCP	Transmission Control Protocol
TOTP	Time-Based One-Time Password
UCS	Universal Character Set
UML	Unified Modeling Language
URL	Uniform Resource Locator
UTC	Universal Time Coordinated (Unofficial)
UTF-8	UCS Transformation Format