Global Information Assurance Certification Paper

Testing stateful web application workflows

GIAC (GWAPT) Gold Certification

Author: András Veres-Szentkirályi, vsza@silentsignal.hu
Advisor: Rob VandenBrink
Accepted: January 11th 2016

Abstract

Most web applications used for complex business operations and/or employing advanced GUI frameworks have stateful functionality. Certain workflows, for example, might require completing certain steps before a transaction is committed, or a request sent by a client-side UI element might need several preceding requests that all contribute to the session state. Most automated tools focus on a request and maybe a redirection, thus completely missing the point in these cases, where resending a request gets ignored by the target application. As a result, while these tools are getting better day by day, using them for testing such execution paths are usually out of the question. Since thorough assessment is cumbersome without such tools, there's progress, but we are far from plug-and-play products. This paper focuses on the capabilities of currently available solutions, demonstrating their pros and cons, along with opportunities for improvement.

1. Introduction

When technology made it possible for web servers to return dynamic content, web applications started out simple. As the development of more and more applications shifted from desktop operating systems to the web, complexity grew. While standards moved at a relatively slow pace, libraries and frameworks filled the gaps and made it possible for web applications to behave like their desktop predecessors. And because of how this "evolution" played out, the underlying infrastructure had been abused in numerous ways – leading to a state where the original semantics had been mangled beyond recognition.

The security industry has been keen to keep up with the web, and many tools were developed to help assessments of web applications. Tools had to adapt, and as HTTP transactions slowly lost their original semantics, more and more manual involvement was needed from the tester (LancerX, 2012). Furthermore, complex web applications contain many entry points, automation is key to perform thorough assessments.

This leaves us with an industry where many products have architectures making testing difficult (Hamiel, Fleischer, Law, Engler, 2011). Unfortunately, these usually handle sensitive enough data that warrants a professional assessment. The main goal is getting results out of web application security testing that exclude false negatives caused by inadequate tooling. To demonstrate the issue at hand, an intentionally vulnerable application was developed that can run without enterprise solutions.

For finding and exploiting these issues, two methods will be evaluated, the first being Burp Suite (PortSwigger Ltd.), the commercial user-friendly GUI tool used by many professionals. The alternative is mitmproxy (Cortesi, 2014) with commix (Stasinopoulos, 2015), two simple FLOSS[1] tools that require some tinkering, but offer advanced functionality in return. The results are similar, the pros and cons of each solution are included for easy comparison.

[1] Free/Libre and Open-Source Software

András Veres-Szentkirályi;vsza@silentsignal.hu

Since the issue of testing stateful web applications is not tied to a specific product, the techniques described can be used in various environments. The toolset used for finding and exploiting vulnerabilities is under active development, but even in the future, the basic ideas are transferable to any similar tools. After all, having problems requiring manual intervention while using automatic tools implies that there is a professional tester capable of applying his/her knowledge to any chosen tool.

2. Security of stateful web applications

2.1. Vulnerable stateful web application

To demonstrate the problem and the solutions in a hands-on manner, it is useful to have an example application to play the part of the "victim". However, most frameworks and their dependencies are complex, so a proof-of-concept application was developed for this purpose using short and simple PHP code. In the hope of making it easy to understand, it does not require additional dependencies to run, and in case of PHP 5.4 or newer, even a webserver is provided by the platform itself (The PHP Group, 2015).

Having Git and PHP CLI installed, the examples below can be followed in a hands-on manner by checking out the repository and running the built-in webserver on an appropriate free port.

```
$ git clone https://github.com/silentsignal/damn-vulnerable-
      stateful-web-app
Cloning into 'damn-vulnerable-stateful-web-app'...
...
$ cd damn-vulnerable-stateful-web-app
$ php -S localhost:5000
PHP 5.6.14-1 Development Server started at ...
Listening on http://localhost:5000
Document root is /.../damn-vulnerable-stateful-web-app
Press Ctrl-C to quit.
```

András Veres-Szentkirályi;vsza@silentsignal.hu

2.1.1. Stateful functionality with independent requests

One of the vulnerable functions of the application is a "ping" functionality to test network connectivity. This involves invoking the `ping` command by calling `system()` with parameters from the user. As it can be seen in the code below, neither validation nor sanitization happens on the input; most experts see the vulnerability just by looking at it.

```
<?php

session_start();

if (isset($_POST['host']) && isset($_SESSION['count'])) {
        header('Content-Type: text/plain');
        system("ping -c $_SESSION[count] $_POST[host]");
        unset($_SESSION['count']);
} else if (isset($_POST['count'])) {
        $_SESSION['count'] = $_POST['count'];
        readfile('ping2.html');
} else {
        readfile('ping1.html');
}

?>
```

The two HTML files (*ping1.html* and *ping2.html*) are just simple forms with one input field each for *count* and *host*, respectively. Using the "ping" functionality involves three HTTP requests. First, the user loads the application (GET) which asks for the packet count. Next, the browser POSTs the packet count, and the application asks for the hostname. Upon the third request, the server executes `ping` with parameters from the request (*host*) and the session (*count*). Below is a diagram that illustrates the three states the application can be in – these are represented by the tree clauses of the `if` statement.

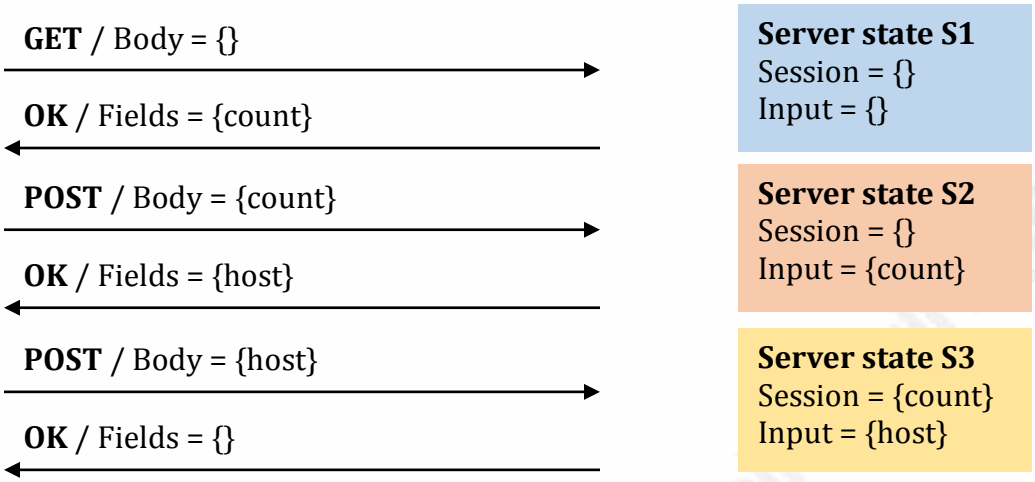


Figure 1. Server states of the stateful "ping" application

Based on the source code and the diagram, it can hardly be a surprise that automatic scanners resending any one of the above requests will fail to detect the issue. Since POST requests inevitably alter the server state, even if the session was in the appropriate state, later requests will arrive to a server in a different state, resulting in false negatives, as it can be seen below on Figure 2. (Two active scans were started on the two POSTs, the single issue is a *Cross-Site Request Forgery*, a typical false positive in Burp Suite.)

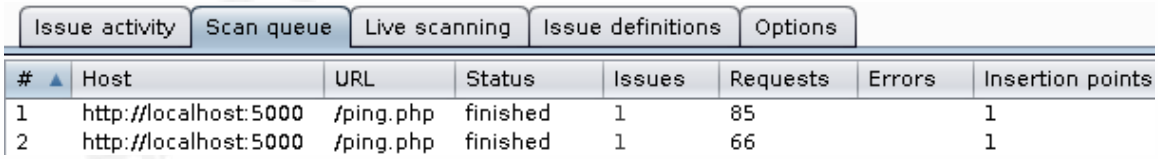


Figure 2. False negatives as a result of naïve usage of *Burp Scanner*

2.1.2. Stateful functionality with requests that depend on previous ones

The other piece of vulnerable functionality of the application is a "traceroute" service to map network nodes. This involves invoking the `traceroute` command by calling `system()` with parameters from the user. As it can be seen in the code below – just like with "ping" – neither validation nor sanitization happens on the input; most experts see the vulnerability just by looking at it.

```php
<?php

session_start();

if (isset($_POST['host']) && isset($_POST['nonce']) &&
    isset($_SESSION['nonce']) && $_SESSION['nonce'] === $_POST['nonce'])
{
    header('Content-Type: text/plain');
    system("traceroute $_POST[host]");
    unset($_SESSION['nonce']);
} else {
    $_SESSION['nonce'] = base64_encode(openssl_random_pseudo_bytes(12));
    echo str_replace('%NONCE%', $_SESSION['nonce'],
                     file_get_contents('traceroute.html'));
}

?>
```

The HTML file is a simple form with a single input field and a hidden one with a placeholder (`%NONCE%`) that gets replaced with the actual nonce value. Using the "traceroute" functionality involves two HTTP requests. First, the user loads the application (GET) which already contains a nonce (also stored in the session) and asks for the hostname. When the browser POSTs the hostname, the server executes `traceroute` with parameters from the request (*host*) after it verified the nonce using the session.

The vulnerability and it being undetectable by naïve scanning is similar to that of the previous "ping" functionality. However, in this case, there is a dependency between the two requests. When sending the hostname in the POST, the body must contain the nonce from the GET response – and by its nature, nonce values are random (Ruby, 2003).

2.2. Using session handling rules in Burp Suite

Burp Suite is often used by professionals testing the security of web applications, and its Professional Edition offers a scanner module to identify issues automatically. By default, the latter resends a single request and modifies it every time in a different manner, while analyzing the response. This works in most cases, when the request leaves the server in the same state as before from the perspective of this test.

However, this is not always true, and Burp provides so-called "session handling rules" to remedy this (Garg, 2013). For example, if the server sends a Set-Cookie header, a naïve scanner would still send the same Cookie headers as before. In this case, the Burp default of using cookies from Burp's cookie jar for the scanner tool solves this. Of

course, these rules are capable of much more, and unfortunately, this results in a complex user interface. But as any complex system, this can also be broken down into simpler parts that are easier to grasp, and combining them gives a powerful tool into the hands of the tester.

2.2.1. Macros

The innermost basic building block for this purpose in Burp is a macro, which can contain one or more HTTP requests. These requests can be independent, which is already helpful if they alter the server state in a way that after the last request, the server is in the right condition. As it can be seen below, when adding a macro, the recorder makes it possible to pick requests from the proxy history. As in the latter module, filters can be used to simplify this process.

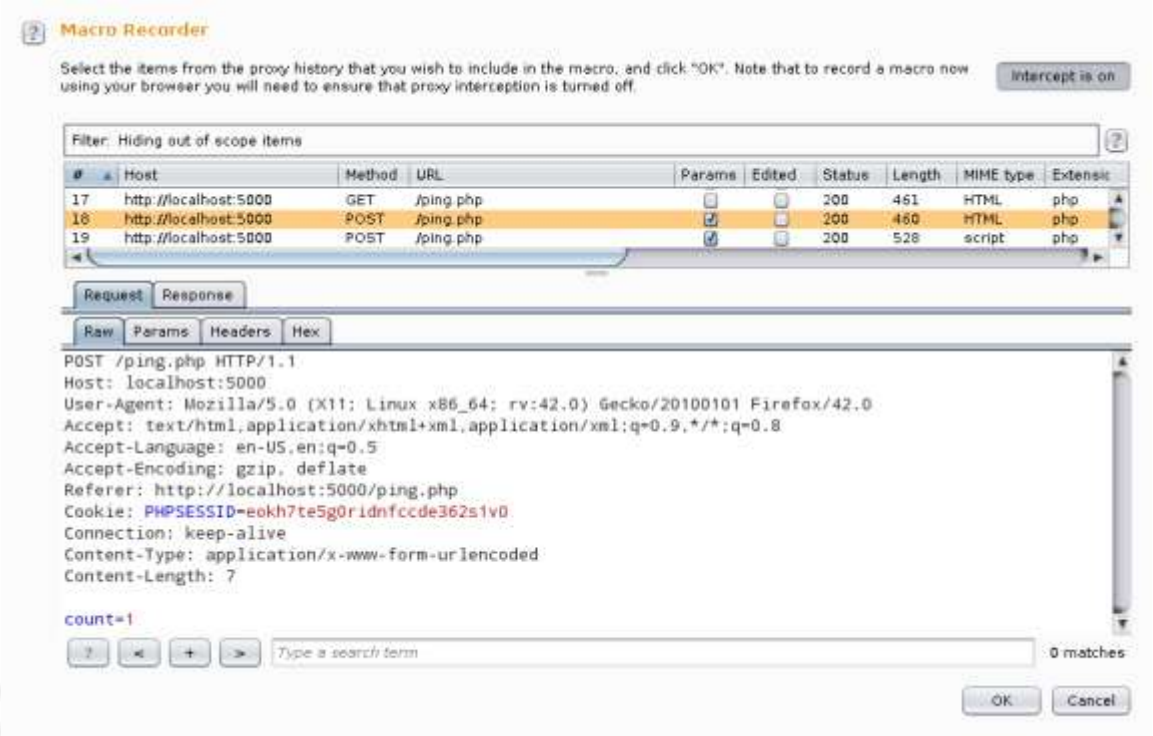


Figure 3. Macro Recorder in Burp with "ping" application requests

In the above example (see Figure 3), setting the ping packet count is the only request that needs to be sent every time before executing the ping command itself – hence it is the only one selected with orange. If more than one request is needed, Ctrl and Shift

can be used to select multiple items or ranges, respectively. Clicking OK in the lower right corner closes the window and finishes the recording process.

After the recording, Burp tries to analyze the selected requests and their relation. This is necessary if the input of some requests depends on the output of preceding requests. While the macro system recognizes some patterns, as with most automated processes, it is not without false negatives and false positives. Fortunately for us, such relations between requests can be fine-tuned manually by selecting a request and clicking on *Configure item*.

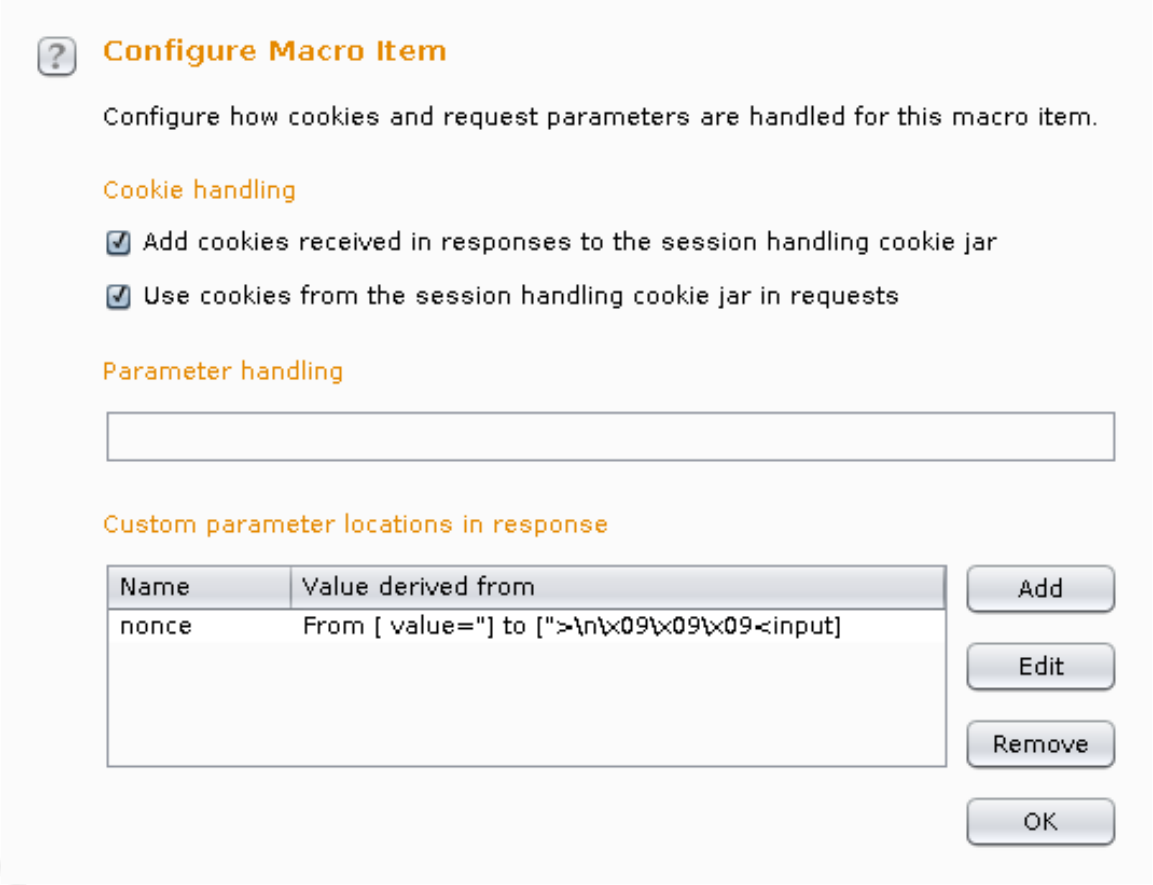


Figure 4. Macro Item configuration window with a custom parameter location defined

In the above example, no parameters were discovered by Burp – although its documentation states it is capable of doing so with these kinds of values. In case of the "traceroute" example, we needed to propagate the nonce to the next request, so a custom parameter was defined. Extraction rules can be specified using several rules, and Burp automatically fills parametric fields based on selecting the value interactively on the UI.

András Veres-Szentkirályi;vsza@silentsignal.hu

As it can be seen below on Figure 5, two algorithms can be used; either defining the start and end (by position or substring) or using *regular expressions*. This might be appropriate for most cases, however, HTML/XML is a *Context-free Grammar*, which is fundamentally more complex than *Regular Expressions* (Gudim, 2014). Even though Burp has the Extender API for plugins, this part of the functionality is not yet included in the interface. This means that there might be some extraction rules that cannot be described by either algorithm below.

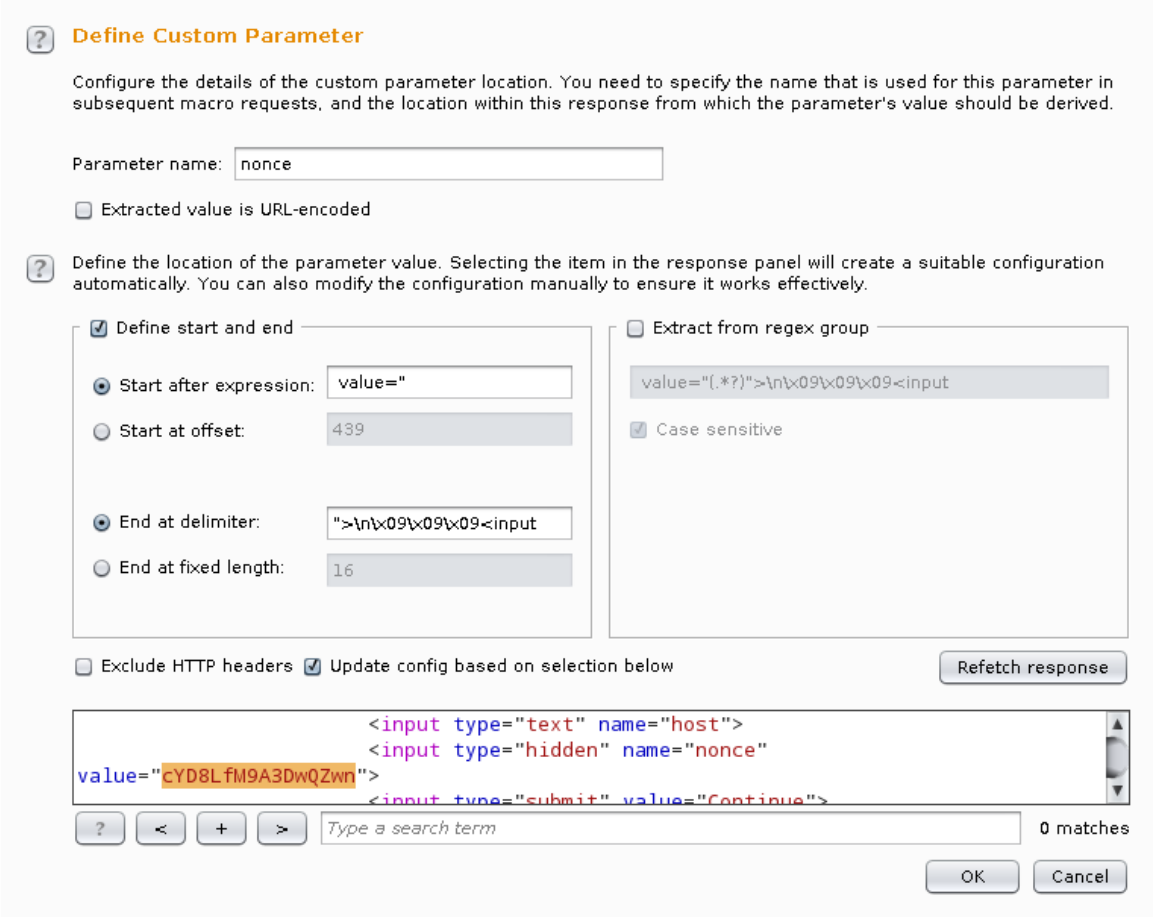


Figure 5. Defining a custom parameter extraction rule in a Burp macro

With the above algorithm defined, the parameter from the response can now be extracted by Burp automatically. This value can be used in later requests within the macro and/or outside the macro as part of a session rule action.

2.2.2. Session rule actions

Once defined, a macro can be assigned to as an action to a session rule. Session rules can run several actions, macros being one of them. The first obvious parameter is the macro itself, and in some cases, this is the only part that needs to be set up. However, if it is not enough that the requests alter the server state, but their output is also crucial for the next request, further configuration is required.

The two ways in which the request could be updated in most cases are parameters and/or cookies. These two can be selected independent from each other, and both can operate on an opt-in (white list) or opt-out (black list) basis. The cookie jar is pretty obvious and does not require much thought – cookies collected during the lifecycle of the macro can be added to the request by Burp automatically.

Parameters are another case, though – these need to be defined within the macro, either the automatic or the manual way. The name used in the macro must match the one used in the request the session rule applies to exactly.

2.2.3. Session rules

Session rules tie a session rule action to a scope of Burp tools (like *Scanner* or *Intruder*), URLs and parameter names. If a request matches the scope, the defined list of actions gets executed by Burp **before** sending it. The defaults are good sane choices, the *Proxy* being excluded from the tool scope being an important example.

The UML class diagram below in Figure 6 helps to understand the relationship between the components discussed above. It is unofficial and by no means complete – but with this number of "moving parts" being defined a complex stack of GUI windows, having a map can help a lot.

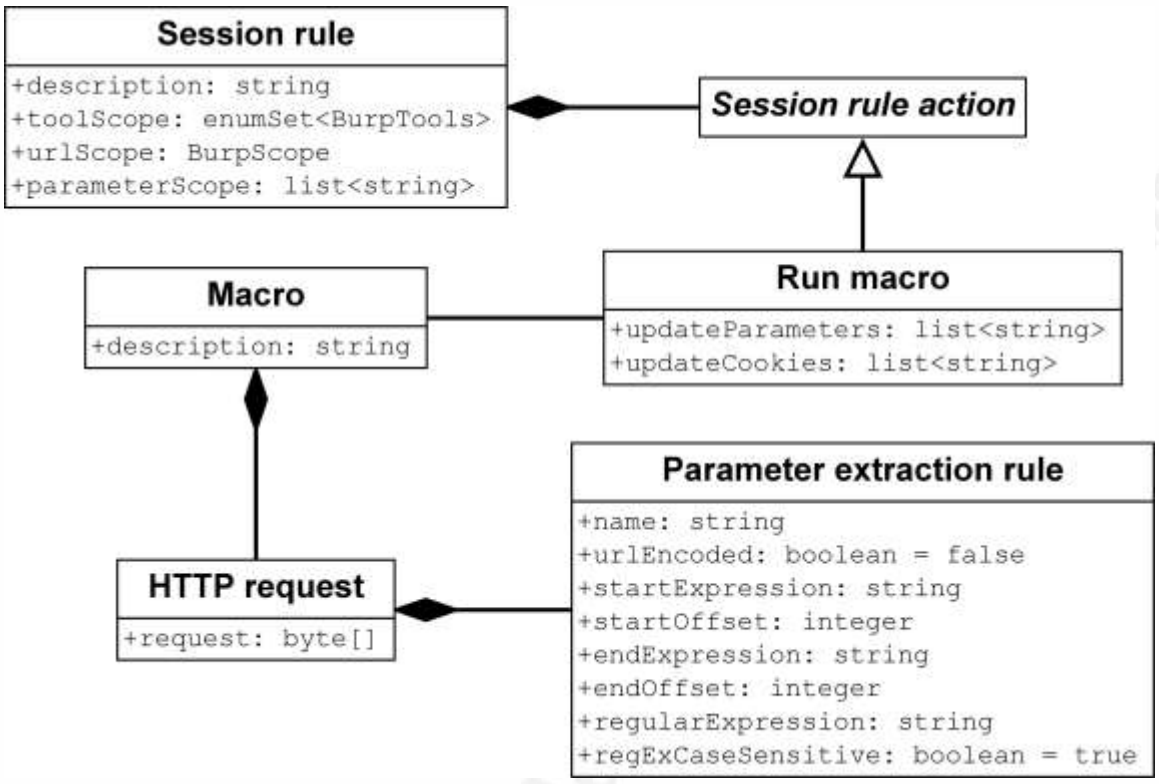


Figure 6. Unofficial UML class diagram of Burp session rules for illustrative purposes

These session rules can be set up in Burp by selecting the *Options* tab, then the *Sessions* subtab. The first section of latter contains the list of current rules, with buttons on the left for managing them. There is another button on the bottom of the section titled *Open sessions tracer*, this can be used for debugging and verification.

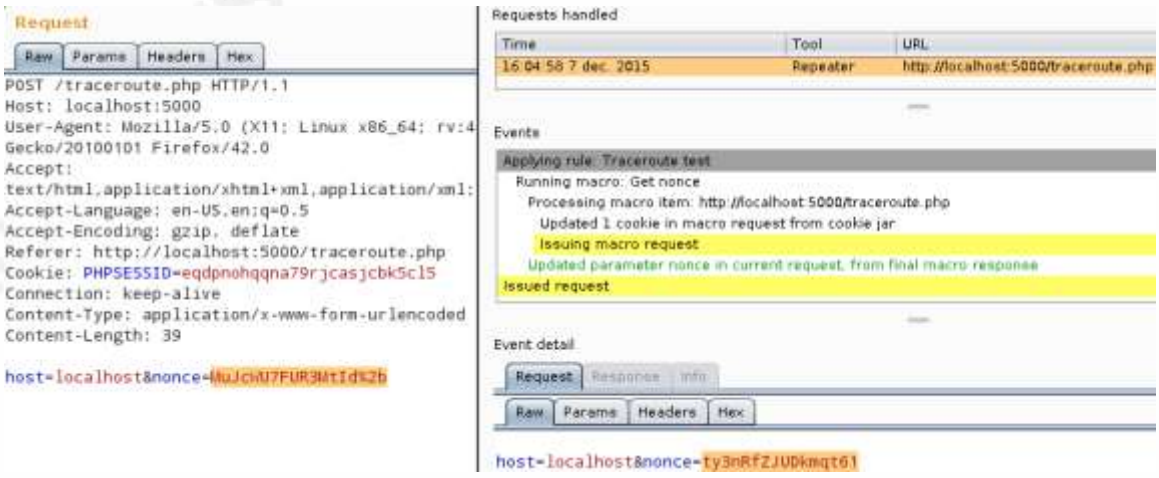
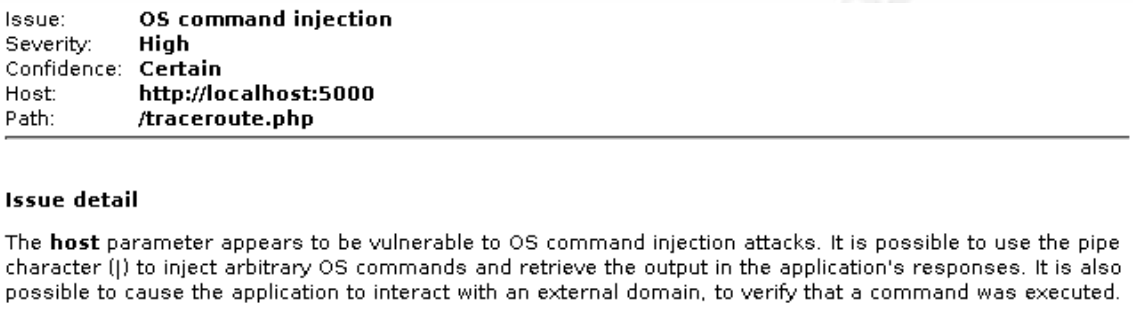


Figure 7. Burp sessions tracer showing verbose trace of the handling of a traceroute request

András Veres-Szentkirályi;vsza@silentsignal.hu

In the example above on Figure 7, the *Repeater* tool issued a request with a "stale" nonce (highlighted with orange on the left). The session manager applied the rule "Traceroute test" and issued a macro request. Based on the response from latter, it updated the parameter "nonce" – the result can be seen on the bottom of the right window, with an updated nonce.

With the session rule in place, the *Scanner* tool quickly found the command injection issue without further manual intervention. As it can be seen below on Figure 8, the parameter "host" and the method of injection was correctly identified by Burp.


Issue: **OS command injection**
Severity: **High**
Confidence: **Certain**
Host: **http://localhost:5000**
Path: **/traceroute.php**

Issue detail

The **host** parameter appears to be vulnerable to OS command injection attacks. It is possible to use the pipe character (|) to inject arbitrary OS commands and retrieve the output in the application's responses. It is also possible to cause the application to interact with an external domain, to verify that a command was executed.

Figure 8. Burp scanner issue displaying a correctly found OS command injection vulnerability

2.3. Using inline scripts in mitmproxy and commix

Mitmproxy is a less-known FLOSS alternative to Burp Suite that offers a character-based interactive UI on the console (Saha, 2014). Its functionality is similar to the *Proxy* and *Repeater* tools of Burp, and can be easily extended using Python modules (Hils, 2015). The availability of the source code makes it possible to edit any part, but inline scripts make modifying requests programmatically even easier.

Since mitmproxy lacks an equivalent of Burp's *Scanner*, external tools need to be used for this purpose. Commix is appropriate for the vulnerable "victim" application, as it is a tool for finding and exploiting command injection vulnerabilities (Stasinopoulos, 2015). These two tools can be easily chained together, as latter supports using an HTTP proxy – which is exactly what mitmproxy provides.

2.3.1. Extracting the nonce using inline scripts

Inline scripts in mitmproxy can be used to provide hooks that are called when specific events occur during the lifecycle of an HTTP transaction. In case of the

"traceroute" application, the script needs to fetch the nonce before the HTTP request is sent, so the `request` hook was chosen. The source code of the inline script can be seen below, followed by analysis and usage information.

```
from urlparse import parse_qsl
from urllib import urlencode
from lxml import etree
import requests

def request(context, flow):
    req = flow.request
    cookies = {key: value[0] for key, value in
                                    req.get_cookies().iteritems()}
    nonce_response = requests.get(req.get_url(), cookies=cookies)
    nonce_tree = etree.fromstring(nonce_response.content,
                                    etree.HTMLParser())
    (nonce,) = nonce_tree.xpath('//input[@name="nonce"]/@value')
    body_items = parse_qsl(req.content)
    for index, (key, value) in enumerate(body_items):
        if key == 'nonce':
            body_items[index] = (key, nonce)
            break
    req.content = urlencode(body_items)
```

From the second, `flow` parameter of the hook, the request is assigned to a local variable called `req` for shorter reference, as it will be used multiple times later. To maintain the session, the cookies used in the original request must be copied to a cookie jar – in this case, a simple dictionary. The Python construct used here is a *dict comprehension*, which build a `dict` object (*map* or *associative array* in other languages) on the fly (Warsaw, 2001). This is needed because the mitmproxy API and the format required by the *requests* library uses a bit different format. Latter is used to issue the HTTP request to fetch the nonce – the same URL is used as in the original request.

Once the response arrives with the nonce, the etree API of the LXML library is used to parse the HTML content (Behnel, 2015). The nonce value then gets extracted from the tree using an XPath expression that looks for the `value` attribute of the `input` node with the `name` "nonce".

Finally, the nonce is updated in the body of the original request. This involves parsing the URL encoded key-value pairs, overwriting the one with the appropriate name ("nonce") and serializing/encoding again in reverse. The body of the original request is

replaced with this new content before sending the request, so server-side checks find that everything is OK.

2.3.2. Vulnerability scanning and exploitation using commix

Commix in itself would not find the vulnerability for the same reasons as the Burp Scanner – as it can be seen below on Figure 9, this results in a false negative.

```
commix ▸ python commix.py --data='host=INJECT_HERE&nonce=khRf20E5hqQxBpIc' --url='http://localhost:5001/traceroute.p
hp' --method=POST --cookie='PHPSESSID=eak63hbuoi9p2mu1tj4va96074'
                                      __
   ___   ___     ___ ___    ___ ___ /\_\    __  _
  /'___\ / __`\ /' __` __`\ /' __` __`\/\ \ /\ \/'\
 /\ \__//\ \L\ \/\ \/\ \/\ \/\ \/\ \/\ \ \ \/>  </
 \ \____\ \____/\ \_\ \_\ \_\ \_\ \_\ \_\ \_\/\_/\_\
  \/____/\/___/  \/_/\/_/\/_/\/_/\/_/\/_/\/_/\//\/_/ ( v0.01b-7f8e86d )

+--
Automated All-in-One OS Command Injection and Exploitation Tool
Copyright (c) 2015 Anastasios Stasinopoulos (@ancst)
+--

(*) Testing the classic injection technique... [ FAILED ]
(*) Testing the eval-based injection technique... [ FAILED ]
(*) Testing the time-based injection technique... [ FAILED ]
(*) Testing the boolean-based injection technique... [ FAILED ]
(x) The 'host=INJECT_HERE&nonce=khRf20E5hqQxBpIc' appear to be not injectable.

(*) The scan has finished successfully!
(*) Results can be found at : '/home/dnet/_projekt/commix/.output/localhost:5001/2015_12_09_16_24_26.txt'
```

Figure 9. False negatives as a result of naïve usage of *commix*

However, when combined with the mitmproxy that has the above inline script loaded, the vulnerability is not only found but can be exploited using an interactive pseudo-terminal. As it can be seen below, commands can be executed easily, with their output printed in a distinctive color.

```
(*) Testing proxy 127.0.0.1:8081 ... [ OK ]
(*) Testing the classic injection technique...
(!) The (POST) 'host=INJECT_HERE&nonce=khRf20E5hqQxBpIc' parameter is vulnerable to Results-based Command Injection.
  (+) Type : Results-based Command Injection
  (+) Technique : Classic Injection Technique
  (+) Parameter : host=INJECT_HERE&nonce=khRf20E5hqQxBpIc
  (+) Payload : !echo 'NUROLT'$(echo 'NUROLT')'NUROLT'

(*) Do you want a Pseudo-Terminal shell? [Y/n] > y

Pseudo-Terminal (type 'q' or use <Ctrl-C> to quit)
Shell > id

uid=1000(dnet) gid=1000(dnet) csoportok=1000(dnet),6(disk),7(lp),20(dialout),24(cdrom),25(floppy),29(audio),30(dip),
44(video),46(plugdev),100(users),108(bluetooth),111(netdev),112(scanner),117(fuse),125(minidlna),141(wireshark),1005
(nexus)

Shell > uname -a

Linux kettohusz 3.16.6 #2 SMP Fri Oct 24 20:54:16 CEST 2014 x86_64 GNU/Linux
```

Figure 10. Finding and exploiting an OS command injection vulnerability with *commix*

3. Conclusion

Burp Suite provided a smooth experience with GUI Windows, typically suitable for power users, who do not necessarily have programming experience. For a point-and-click user interface, it is pretty versatile – at the cost of being deep and complex. The possible algorithms for parameter extraction are simple and cover most cases, without sacrificing performance. Regular expressions consume less resource than full-blown HTML parsing, which has its pros and cons. Having the scanner built-in reduces friction and improves productivity, however, actual exploitation requires redoing much of the work Burp has already done in 3rd party software.

Mitmproxy, on the other hand, provided more of a DIY experience, appealing to hackers with the console UI with less discoverable but more powerful key bindings. Being free in both senses (as in free speech and free beer) results in a different product with more community-driven development, and a shifted focus. The inline scripting interface is directly aimed towards people with Python programming skills, but also offers much broader opportunities at mangling requests. The algorithms used in Burp can be implemented with the built-in string operators and regular expression library of Python manually – but more advanced logic can be implemented as well.

Following the UNIX principle of *doing one thing and doing it well*, mitmproxy also lacks a scanner, thus external tools are required for such tasks. This requires manual "impedance matching" but results in more advanced opportunities. Commix is a good example for command injection vulnerabilities that can work together with mitmproxy, while not only finding issues but making it easy to actively exploit them.

Both the monolithic approach of Burp and the UNIX-y method of mitmproxy with external tools have their merits, and many professionals can use one or the other. But for a real thorough and agile security assessment, one has to combine the best of both worlds, which requires an understanding deep enough to recognize and apply the best solution for the problem.

András Veres-Szentkirályi;vsza@silentsignal.hu

References

Behnel, S. (2015). *The lxml tutorial on XML processing with Python – Parsing from strings and files.* Retrieved 11 December, 2015, from http://lxml.de/tutorial.html#parsing-from-strings-and-files

Cortesi, A. (2014). *mitmproxy – home*. Retrieved 11 December, 2015, from https://mitmproxy.org/

Garg, P. (2013). *Burp Suite Tutorial: Session Handling Mechanisms.* Retrieved 11 December, 2015, from http://resources.infosecinstitute.com/burps-session-handling-mechanisms/

Gudim, V. (2014). *Response to RegEx match open tags except XHTML self-contained tags.* Retrieved 11 December, 2015, from https://stackoverflow.com/a/1758162

Hamiel, N., Fleischer, G., Law, S., Engler, J. (2011). *Challenges in the Automated Testing of Modern Web Applications.* Retrieved 11 December, 2015, from https://media.blackhat.com/bh-us-11/Hamiel/BH_US_11_Hamiel_Smartfuzzing_Web_WP.pdf

Hils, M. (2015). *Inline Scripts – mitmproxy 0.15 documentation.* Retrieved 11 December, 2015, from http://docs.mitmproxy.org/en/stable/scripting/inlinescripts.html

LancerX. (2012). *Testing JSF application with JMeter - ViewState issue.* Retrieved 11 December, 2015, from https://stackoverflow.com/q/12734017

PortSwigger Ltd. (n.d.). *Burp Suite*. Retrieved 11 December, 2015, from https://portswigger.net/burp/

Ruby, S. (2003). *Nonce.* Retrieved 11 December, 2015, from http://www.intertwingly.net/blog/1585.html

Saha, S. (2014). *Reverse-engineering the Kayak app with mitmproxy.* Retrieved 11 December, 2015, from http://www.shubhro.com/2014/12/18/reverse-engineering-kayak-mitmproxy/

Stasinopoulos, A. (2015). Automated All-in-One OS Command Injection and Exploitation Tool. Retrieved 11 December, 2015, from https://github.com/stasinopoulos/commix

András Veres-Szentkirályi;vsza@silentsignal.hu

The PHP Group. (2015). *PHP: Built-in web server – Manual*. Retrieved 11 December, 2015, from https://secure.php.net/manual/en/features.commandline.webserver.php

Warsaw, B. (2001). *PEP 274 – Dict Comprehensions.* Retrieved 11 December, 2015, from https://www.python.org/dev/peps/pep-0274/