hakin9

PRACTICAL PROTECTION

IT SECURITY MAGAZINE

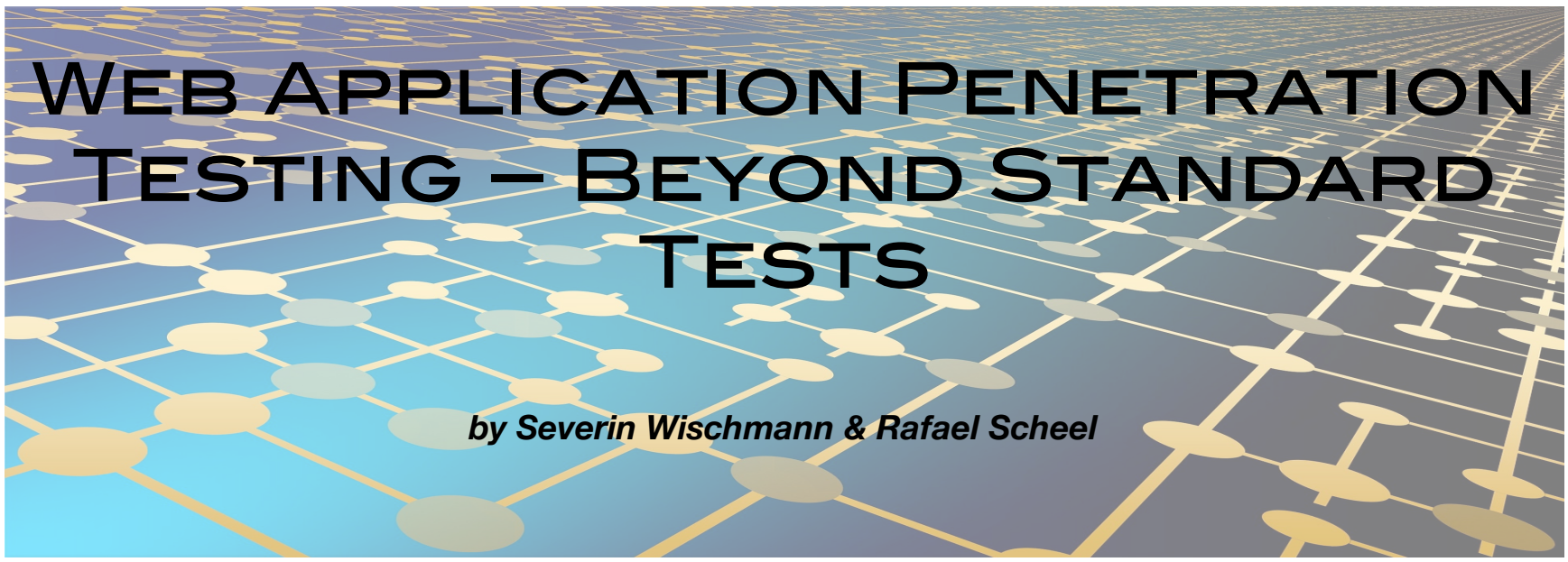
Vol. 11 No. 03

WEB APPLICATION PENETRATION TESTING
Introduction

HOW TO HACK
WITH SQLMAP

SCANNING WEAKNESSES
WITH GOOGLE AND NMPA

WEB APPLICATIONS
THE MODERN WORLD



Web Application Penetration Testing – Beyond Standard Tests

by Severin Wischmann & Rafael Scheel

Watching TV, gathering information, interacting with friends and family or buying groceries: the internet gains more and more functionality by the day. Putting a new service on the web becomes easier every day with simple scripting languages, such as PHP and JavaScript, as well as content management systems (CMS) that provide all the basic functionality. Given these possibilities only one question remains: are they safe to use?

What makes web applications difficult to secure is that an attacker can attack the target on several layers, the operating system, the web server, the executing engine, the application and possibly the transport encryption. So, in theory, and depending on the threat model, one has to secure every OSI layer by itself. And shutting everybody out is not possible as this would reduce the functionality of the web application to zero. This dilemma is then solved by making compromises. An example would be supporting ciphers in cipher block chaining mode (CBC). There are known attacks on encryption algorithms using this mode, so from the security perspective, they should not be allowed. Ciphers without known attacks on them are only present in TLS 1.2, a soon 8 year old standard, which is still not supported by all currently used browsers, e.g. IE 10, and the browser in Android 4.4 has it disabled by default, older browsers do not support it at all. Such compromises should be avoided and, luckily, there are many tools present that detect outdated software and insecure configurations, such as Nessus Vulnerability Scanner[1] or OpenVAS[2]. There are even tools that attempt to test web applications for common vulnerabilities, such as Acunetix[3] or w3af[4]. But especially in the latter case, penetration testing web applications, much like in the game Go, programs are still easily beaten by skilled humans. In the following article we present the process on how to thoroughly test web applications.

Web App Pentest Process

Before starting a penetration test, it is important to collect as much information as possible about the target. This includes details about the network architecture, e.g. are load balancers, (web application) firewalls or reverse proxies in place, can the application be found under different URLs and possibly the operating system (OS), web server and framework name and version.

Then, the first step of a web application penetration test should always include a test of the underlying architecture. The OS, the web server and the used framework to run the web application should be checked for known vulnerabilities and possible security misconfigurations. In terms of the current OWASP Top 10 from

2013, this covers A9 – Using Components with Known Vulnerabilities and parts of A5 – Security Misconfigurations. This part can be automated using a range of tools. Comparing a found version string against a list of version strings with known vulnerabilities is a task a computer is really good at and much faster than a human being.

When the basics are covered, or while some tool is checking the basics, a site map of the web application is needed. This process is also called "Spidering", which means to follow every link on a site to identify the general functionality of the application as well as to discover points of interest, e.g. web forms. This is a task where tool support is readily available, e.g. WebScarab[5] or Dirbuster[6]. It is important that the tools are properly configured, as e.g. hitting a "Logout" link might severely change the outcome or crawling linked sites that belong to other domains might even have legal consequences.

Once the tester is confident that he has found every point of interest, the actual testing of the application's functionality starts. These tests are usually carried out one functionality after another, e.g. completely test the login form before moving on to the contact form.

Finding parameters

Testing a single functionality follows the same principle as the whole penetration test: First, one has to gather all the information possible and then use that information in the test. In the case of a single form, this means to identify all the parameters that could be used to interact with the web application. In our experience, it is especially for more experienced testers more often than not the case that a parameter is not found than that the identified parameters are tested incompletely. Of course, every tester will remember to check all form parameters for injection, SQL, HTML (including Cross-Site Scripting, XSS for short) or otherwise, as most of them are revealed by simple fuzzing[7]. Others are not that widely known and some which are often forgotten are listed in the following.

HTTP parameters: A first additional parameter can be found in the HTTP method with which a form is submitted as GET, POST and PUT. This may lead to different results. Intercepting proxies might offer the functionality to quickly change the method with which the request is sent to the server.

```
POST / HTTP/1.1
Host: duckduckgo.com
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-GB,en;q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
Connection: close
Content-Type: application/x-www-form-urlencoded
Content-Length: 13

q=test+search
```

```
GET /?q=test+search HTTP/1.1
Host: duckduckgo.com
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-GB,en;q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
Connection: close
```

Figure 1: POST request transformed to GET request

Then, the rest of the request header can be included into the set of parameters. Cookies are often a parameter that is not checked for SQL injection, but also, other headers might be parsed by the application and not be properly escaped. One of the parameters most often forgotten is the URL. The

current URL is often included into the HTML that is returned back to the client which again makes it a potential delivery method for XSS attacks. An online shop might perhaps use URLs in the form of "www.online-shop.com/products/books/123456" and include that URL in the response body, but the "/books" part is only there for user guidance and the product is identified only by the trailing ID. Therefore, replacing "books" by any JavaScript code might actually lead to the code being executed in the client's browser.

Forms: The most attractive version of forms are file uploads. To be given the possibility to put chosen content on the server is a penetration tester's dream. In a best case scenario, uploaded code can be executed and the system can be fully compromised. Usually, one has to determine what files can be uploaded, how they are stored and if and how they are processed on the server. A website with image upload capabilities might, for example, enforce the file type by checking the file extension, but not check the content for a matching file type header. In the case of JPG, the server had to check if the file starts with the following two bytes "\FF\D8". This would allow an attacker to upload arbitrary files as long as the last few characters of the file name match. Uploading files provides a number of parameters: the file name, its size and content, the file type one specifies in the request when uploading it and how it is stored on the server (think file system or database).

Of course, file uploads and more general forms on the website are planned interaction points, so they tend to be covered security-wise by the developers.

Guessable parameters: A tester should look further than forms. For example, hidden functionality. As programmers try to name functions concise and logically, a function named "addUser" might indicate that a function named "editUser" exists. The same applies to REST-APIs where "POST" is allowed to create a user on the "/user" route, "PUT" and "DELETE" might be allowed on "/user/<ID>". Another vulnerable spot of web applications are plugins of the used CMS. As they are exposed to less scrutiny, software bugs are more likely to be found there. The names of the used plugins can usually be found in the HTML, some CSS files or the included JavaScript files.

Of course, the retrieval of information has to be tested, too. Being able to access personal information of another user is a severe defect in a web application. These are often linked to a user ID, which should by now already be on the list of parameters to play with. Can the ID be guessed or calculated? Easily guessable IDs are sequential integers. Calculations might be based on the current time or a weak random function, e.g. Java's "java.util.Random" class where all future values can be calculated after knowing two generated numbers in sequence[8].

Testing parameters

After enumerating all parameters, the tests can finally start. As mentioned above, the first tests will probably be checking all form fields for injection by fuzzing. Fuzzing also reveals the range of legal values. This can then be used to further improve sample attacks that exploit potential flaws. Testing a web application is difficult as each one is different. This is where a tester's experience and background really shine.

Programming experience might give useful insights. Imagining how a certain behaviour of an application could be reflected in source code might help finding security holes. A famous example for this is “Drupalgeddon”[9], where a security firm figured out that by passing an array instead of a single variable to the application, one could circumvent the SQL injection filters and therefore, execute arbitrary SQL code.

Experience with data formats can give new insights. An XML parser of a SOAP API that processes external entities might leak information. There is no general advice on how to proceed, as each application is different and has to be tested on its own. One has to just work through the data the application offers and evaluate it for potential risk factors.

Interesting findings

However, this section gives a handful of examples to give penetration testers some hints on how to think, where to look, and what to dig for.

PHP object deserialization

A recent interesting finding was the deserialization of PHP objects in the “User-Agent” header in Joomla, a CMS based on PHP. Due to not properly handling user input, it was possible to gain remote code execution via a serialised PHP object that was injected in the “User-Agent” header. The attack was later improved by moving the attack code to the “X-Forwarded-For” header, as this makes it invisible to standard Apache logs. Keeping in mind that the vulnerability was only detected by looking at server logs, this is a huge improvement.

SQL injection in cookie

While auditing a patient data administration platform in use at several healthcare organisations in Europe, we discovered an SQL injection in the session cookie before authentication. Using this flaw, we were able to access all the patient data and to dump the password hash of the administrative user. Lucky for us, the passwords were hashed using SHA1, which allowed a fairly fast brute-force attack on the hash. This led to the complete compromise of the system.

Custom web server

In another penetration test, Oneconsult found a custom web server implementation. The scope was then extended to include a security code review of the web server. The source code revealed a buffer overflow in the argument collection. When a variable was provided twice, once as a regular variable, once as an array, the buffer length was calculated using the length of the first variable. Exploiting this overflow remote code execution was achieved.

A security code review is a really powerful tool, in general. The tester gains perfect information and can analyse the application’s behaviour on any input. When auditing an application based on open

source software, e.g. WordPress (especially the plugins), this can be a great addition to a test-based audit. Security code reviews take up a considerable amount of resources when done correctly, so some planning is required.

Macro in CSV

While auditing a web application that provides file upload capabilities, our team discovered that *a)* CSV files can be uploaded and *b)* the file will be evaluated by the server. This validated the following request to be sent to the server:

```
POST /secure/file/upload HTTP/1.1
Host: test.host.ch
Connection: keep-alive
Content-Length: 322
Cache-Control: max-age=0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Origin: https://secure.host.ch
Content-Type: application/x-www-form-urlencoded
Accept-Encoding: gzip,deflate
Accept-Language: de-CH,de;q=0.8,en-US;q=0.6,en;q=0.4

Contract=1%2F82563&mutation_per=01.11.1998&number=756.1234.5678.97&personalnr=&lastname=%3Dcmd%7C%27+%2FC+calc%27%21A0&nam
e=Test&daeofbirth=&gender=1&&dokumentensprache=&adresszusatz=&strasse=&postfach=&plz=&ort=&adressland=&csrfPreventionSalt
Request=KZs5InUCvW1YsEIJJ54m
```

Figure 2: CSV with macro code

The highlighted string translates to “cmd /C calc.exe” and was executed on the server.

Aggregation of findings

Sometimes, several smaller findings can be combined to create a more serious vulnerability. One good example would be the “Magento” Remote Code Execution vulnerability that was discovered in January 2015[10]. A similar case, where small vulnerabilities could be combined to something larger, was discovered in one of our audits. The website allowed users to upload their job application, including profile picture and CV. This platform had three flaws that could be combined to imitate the login form or distribute malware through a legitimate URL.

The first thing was that the content-type of an image could be changed to any value. Combining this with the fact that the content was not checked, any file could be uploaded to the server having any content-type. The second thing we discovered was that account pictures of other users could be accessed as long as they were logged in. This is obviously bad for privacy reasons. But you, as an attacker, can send a link to your modified profile picture to any other person. The third vulnerability that was discovered was a persistent XSS in the profile page, which could only be used to include sources from the same page as opposed to sources from anywhere on the internet. Combining these three security flaws will allow the attacker to distribute the link to his application page to anybody on the internet hiding malware in the profile picture that will be accessed as soon as the victim clicks on the link due to the XSS vulnerability.

Conclusion

This article showed that web applications are difficult to build and operate securely as so many details have to be considered and properly addressed. As a penetration tester, the same statement is true. So many details have to be considered and be properly tested. Experience and the constant

thrive to evolve, as the attacks do, is needed to be an asset to a client. Careful development and deployment of a web application are important steps in securing it against outside threats. Penetration tests are helpful to bring a distinct perspective to the development and deployment team and to improve security awareness.

ABOUT THE AUTHORS:

Severin Wischmann works as a penetration tester and IT security teacher at Oneconsult AG. Rafael Scheel is a penetration tester and security researcher at the same company. Oneconsult AG specializes in penetration tests, ISO 27001 security audits and IT forensics.

REFERENCES:

1. Nessus: https://www.tenable.com/products/nessus-vulnerability-scanner
2. OpenVAS: http://www.openvas.org
3. Accunetix WVS: https://www.acunetix.com/vulnerability-scanner/
4. w3af: http://w3af.org/
5. OWASP Project page: https://www.owasp.org/index.php/Category:OWASP_WebScarab_Project
6. Discontinued, but still useful: https://www.owasp.org/index.php/Category:OWASP_DirBuster_Project
7. Fuzzing is a testing method, which tries to reveal errors by inputting random data: http://searchsecurity.techtarget.com/definition/fuzz-testing
8. Here an example on how it could be done: https://jazzy.id.au/2010/09/20/cracking_random_number_generators_part_1.html
9. Official description: https://www.drupal.org/SA-CORE-2014-005
10. Complete write up: http://blog.checkpoint.com/2015/04/20/analyzing-magento-vulnerability/