No Honor Among Thieves: A Large-Scale Analysis of Malicious Web Shells

Oleksii Starov[‡], Johannes Dahse[†], Syed Sharique Ahmad[‡], Thorsten Holz[†], Nick Nikiforakis[‡]

[‡]Stony Brook University
{ostarov, syahmad, nick}@cs.stonybrook.edu

[†]Ruhr-University Bochum
{johannes.dahse, thorsten.holz}@rub.de

ABSTRACT

Web shells are malicious scripts that attackers upload to a compromised web server in order to remotely execute arbitrary commands, maintain their access, and elevate their privileges. Despite their high prevalence in practice and heavy involvement in security breaches, web shells have never been the direct subject of any study. In contrast, web shells have been treated as malicious blackboxes that need to be detected and removed, rather than malicious pieces of software that need to be analyzed and, in detail, understood.

In this paper, we report on the first comprehensive study of web shells. By utilizing different static and dynamic analysis methods, we discover and quantify the visible and invisible features offered by popular malicious shells, and we discuss how attackers can take advantage of these features. For visible features, we find the presence of password bruteforcers, SQL database clients, portscanners, and checks for the presence of security software installed on the compromised server. In terms of invisible features, we find that about half of the analyzed shells contain an authentication mechanism, but this mechanism can be bypassed in a third of the cases. Furthermore, we find that about a third of the analyzed shells perform *homephoning*, i.e., the shells, upon execution, surreptitiously communicate to various third parties with the intent of revealing the location of new shell installations. By setting up honeypots, we quantify the number of third-party attackers benefiting from shell installations and show how an attacker, by merely registering the appropriate domains, can completely take over all installations of specific vulnerable shells.

1. INTRODUCTION

Nowadays, web applications are among the most common attack targets used by adversaries for security breaches. This is likely due to the fact that modern web applications are complex pieces of software and thus they regularly suffer from security vulnerabilities. Coupled with the insight that the underlying web server can be used as a stepping stone into an organization's network, this makes web applications an attractive target for attacks. After a successful compromise, an adversary is thus interested in maintaining a permanent and stealth access to the web server. To this end, she uses a so called *web shell*. A web shell is a piece of software running on a (compromised) web server that provides an adversary remote access to a variety of critical functions (i.e., execution of arbitrary commands, upload and download of arbitrary files, elevation of privileges, or sending spam and spear phishing e-mails). As such, web shells can be seen as a type of *Remote Access Trojan* (RAT) running on a compromised web server, thus being closely related to their counterparts on compromised client systems.

Surprisingly, little is publicly known about the nature of web shells and the surrounding ecosystem. Previous studies of this space treated shells as an artifact of successful attacks and did not analyze them in detail. For example, Canali and Balzarotti found in a large-scale web honeypot study that attackers utilize shells during almost half of the observed attacks [11], but they did not analyze the collected files at all. The practical importance of web shells is further highlighted in a report published by FireEye in August 2015: in this report, the authors noted that web shells are an important building blocks of successful security breaches since they can be used by an attacker during lateral movement in a compromised network [17]. In all of these works, web shells are treated as some kind of blackbox, without providing any insights into their features and potential behaviors.

In this paper, we address this research gap and present the results of a comprehensive study of web shells to shed light into this aspect of cybercrime. To this end, we compile a set consisting of more than 1,400 shells that we use as the starting point for our analysis. By leveraging different data preparation steps, we remove files with shallow differences and create data sets suitable for automated static and dynamic program analysis. In the first analysis step, we uncover the typical features offered by shells to understand which functionality they provide to an adversary. Among other results, we find that (i) many of the analyzed shells leverage some kind of source code obfuscation or cloaking to hide their presence on the web server, (ii) shells typically offer features like password bruteforcers, SQL database clients, and portscanners, beyond the traditional execution of arbitrary commands, and (iii) even the best detection system misses 25% of the shells in our sample set.

In a second analysis step, we are interested in the *invisible* features provided by the analyzed shells. More specifically, we investigate whether shells contain some hidden backdoor or *homephoning* mechanism (i.e., upon execution, a shell surreptitiously communicates to various third parties with the intent of revealing the location of new installations). De-

WWW 2016, April 11–15, 2016, Montréal, Québec, Canada.
ACM 978-1-4503-4143-1/16/04.
http://dx.doi.org/10.1145/2872427.2882992.

tecting backdoors and homephoning mechanisms is a challenging task since we are dealing with complex and potentially obfuscated code, and the attackers have an incentive to hide such mechanisms as well as possible. Hence, we leverage different analysis techniques to reveal such behavior. First, we manually audit a set of almost 500 web shells and find that about 50% of them provide an authentication mechanism in the code. However, we also find that a third of these mechanisms can be bypassed, suggesting that these might in fact be *intentional* authentication bypasses. Second, we utilize honeypots to discover and quantify the client-side and server-side leakage of information since this indicates some kind of potential homephoning. Through the use of multiple honeypot setups, we find that approximately 30% of the analyzed shells exhibit client-side homephoning. On the server side, we find that 4.8% of shells initiate server-side connections to a total of 34 remote IP addresses. During the eight weeks of our honeypot experiment, we observed 690 connection attempts to access the secret URLs of our hosted shells, indicating that backdoors are frequently misused in practice.

Finally, we discover that many shells depend on other domains for remote content, some which had expired and were actually available for registration. By registering the right domains, we show how competing hacking groups and security companies could automatically takeover malicious web shells, or use these domains as a highly accurate method of identifying newly compromised hosts.

In summary, we make the following contributions:

- We collect a set of 1,449 web shells via different methods as a starting point for our research. By means of different data preparation steps (e.g., normalization and deobfuscation), we distill a comprehensive set of shells that represent typical attack tools and we use it for our subsequent analyses.
- We provide novel insights into this type of attack tools by determining the different types of features that shells provide to an adversary. To this end, we utilize both static and dynamic program analysis techniques to discover and quantify the visible and invisible features offered by the web shells we analyzed.
- We experimentally confirm the rumor that many web shells contain backdoors (e.g., authentication bypasses or some kind of homephoning behavior). A comprehensive security analysis based on different analysis techniques such as manual code auditing, automated static program analysis, and execution in a honeypot environment, enables us to discover such backdoors and we provide empirical insights into this behavior.

Data availability: To foster additional research in this area, we plan to make our web shell data set and the analysis results available upon request and proper validation. We hope that the (wider) security community benefits from this information since web shells are a part of the attack landscape that has not yet been analyzed in detail.

2. TECHNICAL BACKGROUND

To provide the necessary technical background to understand the rest of the paper, we present a brief overview of malicious PHP shells and the way that attackers utilize them. Even though a malicious web shell can, in principle, be implemented in any server-supported programming language, we empirically found that PHP, due to its ubiquitous presence on server environments, is the typical language of choice. As such, we will focus on PHP-based shells in the remainder of this paper.

Listing 1 Example of a simple PHP shell

```php
<?php system($_GET['cmd']); ?>
```

A PHP shell is, at its core, a way of executing commands on a remote server. Listing 1 shows the simplest possible shell, receiving commands from a HTTP GET parameter and using the function `system()` to interface with the operating system. The commands are executed with the privileges of either the web server or the owner of the PHP script, depending on the server's configuration.

From an attacker's point of view, the functionality provided by a PHP shell is used to send arbitrary commands to a compromised machine. Malicious PHP shells are typically significantly more complicated than the shell shown in Listing 1. They are often thousands of lines long, belong to different families (*r57* and *c99* being the two most common ones [31]), and make heavy use of obfuscation to avoid detection and make them harder to analyze. In addition to providing command execution on a remote server, they potentially also provide a wide range of other functionality, such as, a file manager, password bruteforcers for FTP and SQL servers, self-remove functionality, and privilege escalation using well-known exploits (see Section 4 for details). Some malicious shells, instead of providing remote control functionality to attackers through a web interface, either turn the remote server into a spam-sending server (or send spear phishing mails from a server within the target's organization), or force it to join a botnet [10].

These malicious web shells are uploaded on compromised web servers through the abuse of all possible vulnerabilities that eventually give the attacker file upload capabilities. These include FTP/SSH credential bruteforcing, abuse of a *remote file inclusion* and *local file inclusion* vulnerability [25,26], *SQL injection* vulnerabilities, or abuse of an existing, benign file-upload mechanism that does not perform proper sanitization of the uploaded files [27].

Next to uploading shells to vulnerable web servers, attackers also seek to abuse existing shells on already compromised web servers. One method of discovering these shells is through the use of search-engine *dorks*, i.e., commands that are meant to return indexed webpages that match specific malicious criteria, such as exposed control panels and pages with sensitive data [15]. For instance, the Google search engine query `filetype:php intext:‘‘!C99Shell v. 1.0 beta’’` is meant to return all indexed instances of a specific version of the c99 shell. Figure 1 shows 6-months worth of visits kindly provided to us by a website designer [36] who has purposefully set up three honeypot pages that use cloaking techniques to pretend to be shell installations to search engine crawlers. The figure shows that, even for a single website, there is a steady stream of tens of attackers, seeking to abuse its already uploaded shells, on a daily basis.

3. DATA COLLECTION

Hundreds of different shells exist on the Web, due to both the presence of multiple versions of the same shell as well as shells that have been used as a base to create new shells. Even though there have been several attempts to collect list of common web shells (e.g. [1–3,5] and more), none of those sources can provide any guarantees regarding completeness

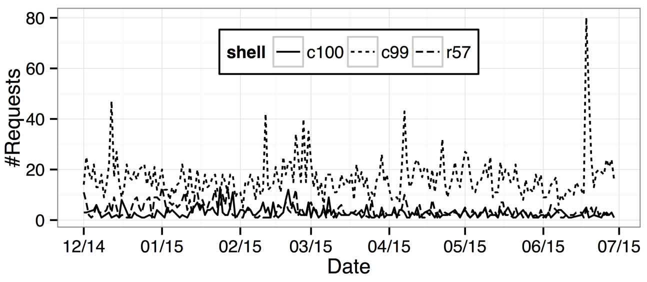

Figure 1: Requests for shells via Google Dorks on one single honeypot domain for a six-month period

and quality. In this section, we thus describe our process for compiling the sets of shells that we used for our experiments.

Data sources and preparation. We started by collecting all shells that we could find in underground hacking websites as well as shells that researchers have observed in their honeypots [1–3,5]. We avoided shells that are designed for legitimate administration of servers since these are outside the scope of our study. By combining all the aforementioned sources, we obtained our starting set of 1,449 shells. Next, we assessed and improved the quality of our working set, filtering-out non-shells (such as files containing only JavaScript code), shells written in languages other than PHP, and shells with shallow differences to avoid potential duplicates. We processed our 1,449 shells as follows:

- *Filtering* was done by checking for PHP tags and a file size exceeding an empirically-determined threshold. Through this process, we excluded 53 files and obtained 1,396 potential PHP shells.
- *Normalization* was performed in order to remove non-obvious duplicates, i.e., shells whose cryptographic hashes may be different, yet are near-identical from a syntactic point of view. Our shell normalization process involved the removal of comments, new lines, whitespaces and semicolons. In addition, we replaced all variable names and function names with a single name. Note that this process is not meant to preserve the correctness of the code. We only use it to cluster shells together and pick the first member of each cluster for further analysis. We empirically found that this simple approach yields sufficient good results to, in practice, detect small differences in different files. Note that more elaborated mechanisms based on an analysis of the *abstract syntax tree*, *fuzzy hashing* [21], or via determining the semantic equivalence [7] could be used in the future, but we found our approach to be sufficient in practice. Using this process, we obtained a set of 804 unique shells.
- *Deobfuscation* was necessary since identical shells identified in the previous step may still be using different obfuscation methods. Hence, we used the state-of-the-art UnPHP deobfuscation service [4] to automatically deobfuscated our set of 804 unique shells. UnPHP returned the deobfuscated code of 661 shells (the service was consistently timing out when trying to deobfuscate the remaining 143 shells), which we normalized once more to arrive at a set of 607 unique shells.

Final datasets. Anyone who has ever attempted to analyze real-world code, especially of a malicious nature, understands that some techniques that may work well in theory might fail to work as expected in practice. Apart from 143 shells which UnPHP was failing to deobfuscate, we also noticed that a number of shells which UnPHP claimed to have successfully deobfuscated, were actually broken. A manual analysis revealed that this was either because UnPHP was not able to correctly escape all special characters in the deobfuscated statements, or because entire pieces of code were missing from the deobfuscated result.

Since automated static analysis and dynamic analysis techniques have different requirements for analyzing code, we decided to create two subsets of our original 607 shells. For the static analysis part, we only used the shells that were either fully syntactically correct or the ones which we could repair with a reasonable amount of manual effort. We also removed shells which we knew would not provide any interesting, from a static analysis point of view, results, such as shells that were just printing local system information or were merely sending an email to a predefined email address. Finally, since we use static analysis to discover popular features of web shells, we removed shells which did not conform to our model of what a shell is, such as shells which were just connecting to a remote server, in a bot-like fashion, and executing any commands which that server would return. At the end of this prefiltering process for static analysis, we arrived at 481 unique shells which we refer in the rest of the paper as the *STATIC_ANALYSIS* set.

For the dynamic analysis part, we employed a more forgiving approach where shells were not removed unless they were completely broken, i.e., we kept the shells whose rendering in a browser would result in some sort of visible UI, even if that was partially broken. This includes the shells that UnPHP was unable to deobfuscate. We removed shells which provided no obvious malicious functionality, such as files calling the `phpinfo()` function and exiting. Finally, we decided to keep shells with bot-like functionality since we wanted to learn more about the nature of the remote servers and attackers. At the end of this process, we arrived at 541 unique shells which we refer in the rest of this paper as the *DYN_ANALYSIS* set of shells.

Our final data set represents different shell families, including variants of *c99*, *r57*, *WSO*, *B347k*, *NST*, *NCC*, and *Crystal*. However, many variants evolved from a family sibling to a new shell family by applying different code obfuscation techniques, adding new features, or copying code from other shell families. For example, the *c99* shell was extended by a privilege escalation feature and renamed to *c999* shell and a fraction of the *c99* shell's code can be found in the *Fx29* shell family. PHP shells that are not explicitly labeled in the source code are thus hard to classify and would lead to inaccurate results.

4. ANALYSIS OF SHELL FEATURES

PHP shells evolved from simple one-liners to complex, multi-functional web applications that provide the attacker with a large set of features that go beyond the execution of a single system command. Since little is know about the actual features offered by such shells, our goal is to enumerate the different types of features that PHP shells provide to the attacker and to identify the most popular ones. This provides insights into the attacker's needs and desired steps after a web server is compromised. At the same time, we enumerate the number of shells that implement features in order to hide the presence and activities of the shell.

4.1 Stealthiness

It is in the attacker's interest to hide the presence of a malicious backdoor in order to keep server access as long as possible. Thus, the source code of many PHP shells is heavily obfuscated since this approach gives the attacker a variety of advantages. First, an attacker aims to hide the shell's intention and capabilities in case it is found by an administrator of the compromised server. Second, obfuscation can hamper the automatic detection by static analysis tools, search patterns, or antivirus products (see Section 4.3). Another reason for obfuscation can be to hide the presence of a backdoor or homephoning mechanism (see Section 5) within the PHP shell from being detected by another attacker. We analyzed our final set of 481 shells in the *STATIC_ANALYSIS* set for signs of obfuscation and compared the original scripts with their deobfuscated versions. This analysis led to the following two obfuscation statistics:

- 20.6% (i.e., 99 shells out of 481) have an increased token count (more than 1%) when enumerating the number of tokens with PHP's internal tokenizer after deobfuscation, indicating unpacked PHP code.
- 20.8% (i.e., 100 shells out of 481) use `eval()` with an average of 15.2 calls to `eval()`, reaching up to 91 calls.

Next to obfuscation that ensures server-side stealthiness, we encountered techniques to hide the presence of the shell from curious client-side visitors. This behavior is likely based on the following insights: since search engine bots crawl and index the entire web, they are likely to also index uploaded shells that were left unprotected. This will then allow other attackers to discover already compromised servers (through the use of the aforementioned search-engine *dorks*) and take over the uploaded shells. In addition, some search engines now report that a website is most likely compromised, as part of their search results. This likely leads to faster cleanups which is beneficial for website owners but not for attackers. For these reasons, shell developers implement basic techniques that hide the presence of the shell from search engine crawlers. Based on the HTTP `User-Agent` header, a request of a crawler can be identified in a rather straightforward way and a different behavior can be emulated. In our deobfuscated set of 481 shells, we identified 76 shells (16%) that check the user agent against a blacklist. For example, 61 shells respond with a *404 Not Found* page if the user agent contains the keyword *Google*. Furthermore, we identified 11 shells that respond with a *404 Not Found* page by default, unless the visitor sends the correct login credentials that will reveal the shell's presence and features.

In summary, we empirically find that 15-20% of the analyzed web shells leverage hiding techniques to conceal their presence on a compromised web server, both from the owner of the web server, as well as specific client-side visitors.

4.2 Interface Features

To better understand and enumerate the features that web shells make available to attackers, we analyzed our *STATIC_ANALYSIS* set of shells with static taint analysis techniques. For this purpose, we extended an existing tool of our design [14] that is able to detect security vulnerabilities in PHP applications, such as *remote command execution*, *remote code execution*, *mail header injection*, and *SQL injection* vulnerabilities. Furthermore, the tool supports the analysis of file-related vulnerabilities, such as *file upload*, *arbitrary file write/create*, *file disclosure*, *permission manipulation*, or *file inclusion* vulnerabilities. Each of these vulnerabilities is interpreted as a feature when detected in the deobfuscated PHP shell's code by our prototype.

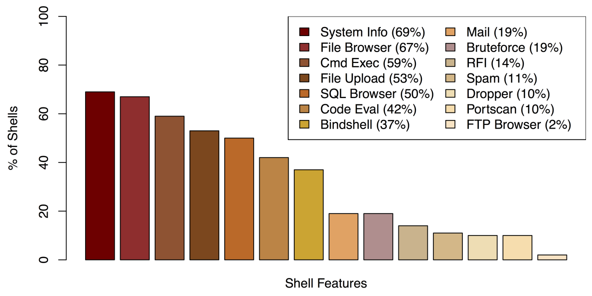

Figure 2: Feature distribution of the analyzed shells

Our static analysis tool also performs context-sensitive string analysis. This allows us to automatically inspect the markup that reaches a sensitive sink and to enumerate further features it cannot detect out of the box. For example, when our tool analyzes sensitive sinks that execute system commands, the scanner reconstructs all possible strings (i.e., commands) that reach this sink. At this point we applied regular expressions to identify commands that we found commonly related to a *file dropper* (`wget, curl, lynx, get, fetch`), a *reverse* or *back connect shell* (`perl, python, gcc, chmod, nohup, nc`), or *information gathering* (`uname, id, ver, sysctl, whoami, $OSTYPE, pwd`). The results were complemented by additional feature detection algorithms that we added to our prototype. For example, when a download of a remote file through PHP's built-in file features is requested, a *file dropper* feature is reported. Likewise, when the output of PHP's built-in functions or reserved constants related to system information (e. g., `php_uname()`) is encountered during taint analysis of a sensitive sink that prints data to the HTML response page, a *system information gathering* feature is detected.

Moreover, we used the annotations regarding loops in the control flow graph representation of a given shell in order to classify certain vulnerabilities as features. For example, a *mail header injection* within a loop is interpreted as a *spam* feature and a login attempt to an FTP server or an HTTP basic authentication within a loop is interpreted as a *brute-force* feature. Similarly, when a built-in function is used within a loop to establish a socket connection and the port is the subject of iteration, a *portscan* feature is logged. Due to space limitations, we omit all (inter-procedural) implementation details that we added to the tool.

The final results of our feature enumeration is shown in Figure 2. The most prominent feature, appearing in 69% of all shells in our collection, is the gathering of system information. More specifically, the current working directory, operating system, and the PHP version is of interest to the attacker. Next, the interaction with the file system is supported by 67% of the analyzed shells. We summarized detected vulnerabilities regarding file read (54%), create (54%), list (40%), delete (38%), edit (29%), and modification of permissions (22%) to one feature named *file browser*. Separately, we found a file upload feature in 54% of the analyzed PHP shells. Next to a file browser, we detected a slightly less popular *SQL browser* to list all databases and tables in about half of our shells, as well as rarely available *FTP browsers* in only 13 of our 481 analyzed shells.

The traditional *command execution* feature was detected in 59% of our shells. Next to arbitrary OS command execution, the shells often provide a prefixed list of commands in the web interface, i. e., the *c99* shell proposes commands to find configuration and password files, or writable directories. Less frequently offered (43%) is the ability to execute arbitrary PHP code through `eval()` and similar operators. This feature appears in smaller shells that focus on stealthiness rather than feature completeness. Additionally, we found 68 shells that offer arbitrary code execution through *remote file inclusion*. In order to bypass restrictive firewalls, remote command execution can also be bound to a separate port that is opened through an external program (37%), preferably written in *Perl* or *C*.

Other features are less frequently available and therefore likely less important for an attacker as a core feature of a PHP shell. For example, only 19% of the shells allow to send an email and 12% of the shells allow to send out multiple emails (which likely represents spam). The feature to launch bruteforce attacks against FTP and HTTP authentication credentials (19%) or port scans (10%) are also rarely available in practice. Note that these tasks can be performed independently from the attacked server. However, when used as part of the PHP shell, the attacker is able to hide her IP address while the compromised web server acts as a proxy.

Since our feature enumeration is based on code-fingerprints, it could, in principle, fail for real-world PHP applications. However, the code of PHP shells is comparably simple (no OOP features, minimum inter-procedural analysis necessary) and we did not encounter false positives during a manual investigation of 50 sample shells. Nonetheless, we found further features that were left out of our evaluation, for example, the ability to crack password hashes, attempts to bypass PHP's `safe_mode`, a HTTP proxy, and *Denial of Service* (DoS) features. We expect these features to be less prominent in PHP shells and hard to fingerprint generically, thus, we decided to not include these into our evaluation.

4.3 Bypassing Antivirus Engines

Antivirus software—despite all its known limitations—is arguably among the most popular security software among users. Next to user and corporate environments, many web servers utilize antivirus software to detect possibly malicious uploads to their servers, either through the use of a legitimate upload functionality (e.g., attachment in a webmail application) or through the abuse of file-uploading vulnerability (such as a *remote file inclusion* vulnerability).

To gauge the ability of modern antivirus software to detect malicious web shells, we uploaded all 481 shells of our *STATIC_ANALYSIS* set to VirusTotal [9], an online service which, at the time of this writing, uses 56 antivirus engines to scan uploads and URLs. For 90% of our shells, at least one antivirus engine of VirusTotal was able to detect a shell as malware, with each shell being categorized as malware by, on average, 15 scanners. The two most accurate antivirus engines could detect 72.5% and 67.2% of our uploaded shells, respectively. These numbers indicate that, even though a collection of antivirus engines could detect most of the shells, even the single most precise engine (Avast) cannot detect about 25% of malicious web shells.

In an effort to understand the effect of obfuscation on the detection of malicious web shells, we also uploaded the deobfuscated versions of the same 481 shells to VirusTotal. Surprisingly, the overall accuracy decreased to 88.5% of shells being detected by at least one engine, and each shell now being detectable by, on average, only 10 scanners (compared to the average of 15 scanners for their original obfuscated versions). As an example, one specific shell that was detectable by 22 antivirus engines became detectable by only two engines when we used its deobfuscated version. This decrease in the detection of malicious shells indicates that some antivirus engines are most likely hardcoding pieces of the original obfuscated shell in their database, without ever deobfuscating the malicious web shell.

Listing 2 Example authentication mechanism.

```php
<?php
session_start();
// configuration
$md5 = '';
// authentication check
if(!isset($_SESSION['login']))
  if(empty($md5) || (isset($_POST['pass'])
  && (md5($_POST['pass']) == $md5)))
        $_SESSION['login'] = true;
  else { die("404␣Not␣Found"); }
```

5. BACKDOORS IN WEB SHELLS

Since web shells are complex web applications, typically measuring thousands of lines of code and using multiple obfuscation layers, it is not unreasonable to assume that some of these shells may be backdoored by attackers who decided to capitalize on them. In the context of web shells, we differentiate between traditional backdoors that involve a bypass of an authentication mechanism, and "homephoning" backdoors where a shell, upon loading, broadcasts its location to third parties other than the attacker who uploaded the web shell to a vulnerable web server.

5.1 Authentication Bypass

We manually examined our set of 481 deobfuscated shells (*STATIC_ANALYSIS* set) for authentication mechanisms that restrict the shell access to users that possess a secret (e.g., a password). At the same time, we audited these mechanisms for possible bypasses. Our intuition is that malicious attackers do not freely publish their shells as a service to "fellow" attackers, but rather to gain access to servers that are compromised by uncautious attackers using that shell.

Most of the authentication mechanisms found were simply based on username/password credentials, either supplied by HTTP basic authentication or an HTTP POST request via a HTML login form. Furthermore, access was limited to a given IP address or IP address range. We also observed samples that require a secret key supplied via hidden HTTP GET parameter or user agent, as well as samples that expect a password which is then used as a XOR key to decrypt the evaluated PHP code. Most of the detected mechanisms, however, default to no required authentication, for example, if the password in the configuration code is left empty.

Listing 2 shows an authentication mechanism that is used in 23 of our analyzed PHP shells. Here, the variable `$md5` is left empty and the session key *login* is automatically set to `true` which authenticates the user's session. If the variable `$md5` is initialized with the MD5 hash of a password, however, the user has to provide the correct password for authentication via an HTTP POST request before he can access the shell's features. Thus, by setting a password in the shell's code, the authentication mechanism is activated.

Listing 3 Variants for simulating `register_globals`.

```
<?php
extract($_REQUEST["c99shcook"]);
parse_str($_SERVER['HTTP_REFERER']);
import_request_variables("GPC");
foreach($_GET as $k => $v) { $$k = $v;}
```

In order to automatically enumerate activated and deactivated authentication mechanisms, we installed our set of shells in a sandbox and visited their root page. Based on our static and dynamic analysis, the following results were observed for our set of PHP shells:

- 52.0% provide an authentication mechanism in the code, i. e., 250 out of 481
- 30.8% of the authentication mechanism can be bypassed, i.e., 77 out of 250
- 28.4% of the authentication mechanism are activated by default, i.e., 71 out of 250
- 25.4% of the, by default, activated authentication mechanism can be bypassed, i.e., 18 out of 71

In the following, we present different types of backdoors we encountered in our set of PHP shells that can be used to bypass the authentication mechanism. We believe that these were installed intentionally by the creator or a redistributor of the PHP shell. We also observed that many authentication mechanisms were copied among different shell families and variants, including (perhaps unintentionally) the backdoor code. The detected backdoors can be grouped into three categories: (i) registering global variables, (ii) using unprotected features, and (iii) leaking the authentication credentials. Note that, due to the code complexity and large number of analyzed PHP shells, we do not claim completeness for the detection of all backdoors in our set.

5.1.1 Register Globals

The PHP setting `register_globals` was activated by default before PHP version 4.2.0 and introduced many security issues [29]. The setting allows an attacker to initialize any global variable via HTTP request parameter. This led to unexpected behavior and security issues, for example, when the GET request to `index.php?loggedin=1` initializes the variable `$loggedin` in *index.php* to the value *1*. Thus, the `register_globals` directive was deactivated by default in later PHP versions and was removed as of PHP 5.4.0.

Still, the same behavior can be achieved by using PHP built-in features, as shown in Listing 3. For example, the built-in functions `extract()` and `parse_str()` can be used to populate values in an array or URL string into the global scope. When these functions are called with user input and no additional arguments, the `register_globals` directive is, effectively, simulated. The same applies to a call to the built-in function `import_request_variables()`. Moreover, each key/value pair within the superglobal `$_GET` or `$_POST` array can be populated into global variables by constructing a loop and assigning each key to a variable dynamically.

The security implications of these one-liners are subtle and hard to spot for untrained eyes, making them a perfect choice for planting backdoors in PHP shells. In fact, in 70.1% of the backdoored PHP shells, one of these features was injected between the authentication configuration and the check, allowing to bypass the authentication completely. For example, the authentication mechanism shown in Listing 2, when activated and backdoored, could then be bypassed by overwriting the variable `$md5` with an arbitrary password or by setting the `$_SESSION['login']` key directly (`index.php?md5=0&_SESSION[login]=1`). This backdoor is found in every *c99* shell and was copied to a variety of sibling shells that adopted *c99*'s extensive configuration code that includes a subtle call to `extract()`.

Listing 4 Example authentication with backdoor locations.

```
// position 1
$s_auth = false;
if(is_authenticated()) { $s_auth = true; }
// position 2
if($s_auth) { // protected shell features }
// position 3
```

5.1.2 Unprotected Features

Other authentication mechanisms could be bypassed during our analysis by abusing unprotected functionality (22.1%). Sometimes likely by accident, sometimes clearly intentional, certain features are not protected by the authentication mechanism and can be abused by an insider. Listing 4 demonstrates three positions within a protected PHP shell where we found backdoor code. For example, we found several code execution vulnerabilities at position one or two, which are accessible before the authentication is performed. This allows an attacker to either upload another shell to the compromised server or to retrieve the shell's source code including its login credentials. Moreover, one shell switched the variable `$s_auth` to true in case the GET parameter *error* was set. Other samples were extended with a new feature at position two or three. Although the corresponding HTML interface was not visible without authentication, the backend of these features could still be accessed and abused by an insider. For example, the feature to download the current directory as a ZIP file allowed us to also retrieve the PHP shell itself, revealing its source code and login credentials.

5.1.3 Information Leakage

Last but not least, some shells allow the circumvention of the authentication mechanism by leaking the authentication credentials. For example, in seven shells, an email is sent to the attacker's mailbox that includes the compromised server's domain, path to the shell, as well as the authentication password. The original code was obfuscated and hidden within the shell's features. This makes it hard to spot this functionality during a code review, specifically when only the authentication mechanism itself is investigated.

5.2 Homephoning

Since web shells are typical web applications where attackers communicate with the compromised servers via their web browsers, shells can leak their location or other kinds of information both from the client-side and the server-side.

At the client-side, when an attacker loads the page in her browser, JavaScript code can be used to communicate to the outside world and purposefully leak the URL of the current page, i.e., the shell's URL, by leaking the value of the `document.location.href` browser property. Moreover, all browser requests for remote resources, such as images, Cascading Style Sheets, and JavaScript scripts, will typically include the current page's URL in the `Referer` header of each request. This allows not only for intentional homephoning, but also unintentional one, e.g., a web shell leaking its location to Google, as it is fetching an analytics script.

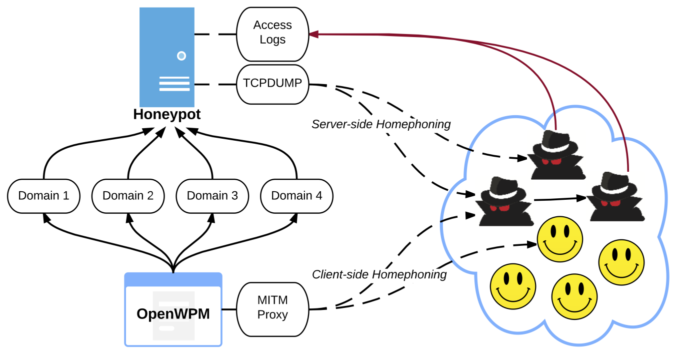

Figure 3: Honeypot architecture for measuring web shells homephoning

At the server-side, the shell can communicate with the outside world through the use of a wide range of functions provided by the programming framework itself, or by interfacing with the underlying operating system. For instance, in PHP, a malicious shell can communicate with third parties by using sockets, HTTP APIs (e.g., PHP's `http_get()` function), or using the ubiquitous `wget` utility.

Setup

To discover and quantify the client-side and server-side leakage of malicious web shells, we set up a honeypot infrastructure depicted in Figure 3. The details of this setup, involving our set of 541 shells (*DYN_ANALYSIS* set), are as follows:

1. Whenever a shell required explicit authentication, we modified the underlying code to remove this requirement. This allowed us to reduce the complexity of our overall setup since we can later use a shell by merely requesting the appropriate URL. These shells were then copied to an Amazon EC2 virtual machine, in a non-web-accessible folder.

2. We registered four available domains and resolved their IP address to our virtual machine. The keywords included in our four domains belong to four different categories, namely: sports, hacking, shopping, and banking. We chose to register domains indicating different websites in an effort to entice as many attackers as possible, who would see our domain names in the potential requests of their homephoning shells.

3. We utilized the OpenWPM framework [16] on a second server, to emulate an attacker browsing through each of the uploaded shells through one of the four registered domains. While OpenWPM is intended for web privacy measurements, its ability to record all client-side, third-party requests was useful in later identifying intentional and accidental client-side homephoning.

4. We implemented a series of scripts that allowed the OpenWPM framework to synchronize with our server. Namely, as the pseudocode in Algorithm 1 shows, OpenWPM announces its intention to visit a specific shell, causing the server-side to make that shell available through one of our four registered domains and a specific file path. For instance, if OpenWPM indicated that it wants the URL for `shell3.php`, our server-side code would produce a link such as `www.banking.com/img/lib/shell3.php`. The filepath not only makes our overall URLs more believable for anybody watching, but also allows us to later differentiate between generic crawlers visiting our main domains, and attackers who know, because of homephoning, the exact path to their malicious shell. After waiting for a predetermined number of seconds, our client signals the server to remove the analyzed shell from our web server's root directory.

Algorithm 1 Pseudocode of the synchronization between client and server, for the detection of shell homephoning

```
▷ Server-side code
function NEXTSHELL(shell)
    $current\_domain \leftarrow$ PICKRANDOMHONEYPOTDOMAIN()
    $current\_path \leftarrow$ PICKRANDOMHONEYPOTPATH()
    COPYTOWEBROOT($current\_domain, current\_path, shell$)
    STARTTCPDUMP()
    return $current\_domain + current\_path + shell$
end function

function DELETESHELL(shell,shell_url)
    $current\_trace \leftarrow$ STOPTCPDUMP()
    STORETRACE($current\_trace, current\_date, shell$)
    REMOVEFROMWEBROOT($shell, shell\_url$)
end function

▷ Client-side code
for $shell$ in $shells$ do
    $shell\_URL \leftarrow$ NEXTSHELL($shell$)
    OPENWPM($shell\_URL$)
    SLEEPSECONDS(30)
    DELETESHELL($shell, shell\_URL$)
end for
```

The set of 541 shells was visited three times a day, for a period of 8 weeks, starting from May 24, 2015. The only assumption that we made in this experiment is that, if a shell contains homephoning code, the execution of that code should not be hard to trigger. That is, we assume that a homephoning shell will greedily contact various predetermined remote hosts upon the mere loading of its main page, rather than wait for the unsuspicious shell user to click on some specific link of the shell's UI.

Results

Client-side homephoning: By inspecting the logs of client-side, third-party requests, as captured by the OpenWPM platform, we discovered that 29.2% of PHP shells contact third-party domains, with an average of two domains contacted per shell. Overall, our set of shells contacted a total of 149 domains, resolving to 108 unique IP addresses. Table 1 shows the 15 most contacted domains as well as our manually curated categorization of these domains. One can notice that web shells initiate third-party requests towards a wide range of remote hosts, some of which are clearly benign. These are cases of *accidental* homephoning, where a shell, in the process of including a remote object, inadvertently leaks its location to the remote host.

Server-side homephoning: The set of steps described in Algorithm 1 allow us to precisely determine client-side homephoning, as well as maintain an always available web server whose logs we can later analyze to determine whether attackers, notified by their homephoning shells, are trying to access our server. This setup, however, falls short of allowing us to do proper attribution of server-side homephoning of specific shells. This is because, server-side processes that

Table 1: Top 15 Client-side Homephoning Targets

Domain	#Shells	Description
alturks.com	43	Parked domain
w0rms.com	20	Hackers portal
jino.ji.funpic.org	9	Under construction
front.facetz.net	6	RU analytics
hit4.hotlog.ru	6	RU analytics
[...].pp.regruhosting.ru	6	Not found
w.uptolike.com	6	RU analytics
sync.audtd.com	6	RU analytics
display.intencysrv.com	6	RU analytics
cm.g.doubleclick.net	6	Ad services
counter.yadro.ru	6	Used by adware
sync2.audtd.com	6	RU analytics
fonts.googleapis.com	5	Google APIs
www.fbvideo.16mb.com	4	Suspicious
data.t00ls.org	4	Active attacker

are initiated by any given shell, can outlive the deletion of the shell, as that happens in the NEXTSHELL step.

To account for this behavior, in addition to our long-lived 8-week experiment, we also run another experiment where we revert to a known clean copy of the remote virtual machine after the analysis of each individual shell. We use the results of this experiment to do proper, server-side homephoning attribution without worrying about the attacker's ability to connect back to our honeypot service.

By inspecting our collected packet traces and filtering out connections due to Ubuntu update checks, DNS servers, and Amazon-EC2-specific traffic, we discovered that 4.8% of shells initiate server-side connections to a total of 34 remote IP addresses. The remote hosts are located all across the world with the top three countries being USA (16), Republic of Korea (4) and China (2). Moreover, we note that 70% of the server-side homephoning shells connect to specific domains, instead of using hardcoded IP addresses. Specifically, we detected 21 unique domains being contacted, not counting ones that were just resolved but never actually contacted. The contacted domains include clearly benign destinations like `google.com` and `nmap.org`, IRC destinations like `irc.dal.net` and `irc.studentwine.co.uk` for botnet communication, and obviously suspicious domains such as `pc117.zz.ha.cn` or `www.weigongkai.com`.

Connecting attackers: During the eight weeks of our honeypot experiment, we received 690 attempts to access the URLs of hosted shells, from 71 unique IP addresses, located in 17 countries with the top three being Turkey, USA, and Germany. As mentioned earlier, since these URLs, e.g., `www.banking.com/img/lib/shell3.php`, were never made public, anyone who knows them, *must* know them because a shell, either through client-side, or server-side homephoning, leaked its precise URL to an attacker.

These 690 requests were targeting 30 of our 541 monitored shells, showing that not all homephoning shells will eventually be accessed by attackers. This could be either because of accidental homephoning, where the recipients do not know that shells are contacting them, or because an attacker is no longer present to react to homephoning requests. In Section 7 we show how competing attackers or defenders can take advantage of this behavior to hijack web shells.

Figure 5 shows the time series of the requests received per day for our honeypot web shells. Even though the number of daily requests varies significantly, the smoothed average shows a steady increase of the requests until roughly the middle of our monitored period, followed by steady decreased till the end of our 8-week experiment. This behavior

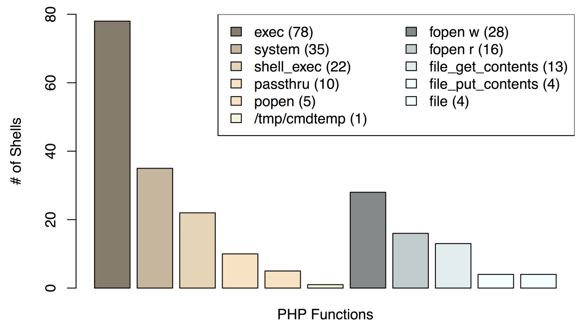

Figure 4: Number of shells that use particular PHP commands silently

Table 2: Files that Shells silently access or modify

Group	#Shells	Examples
/var/www/html/∗/∗	21	write test.txt, read own PHP file
/etc/∗	8	named.conf, hosts, passwd, fstab
http://∗	7	send own URL, load api.php, sender.txt
∗.php	6	dbs.php, shell.php, errors.php, ss.php, etc.
/tmp/∗	5	qw7_sess, shellcode.so, Ra1NX
php.ini	5	write to php.ini or ini.php
.htaccess	5	write .htaccess or sym/.htaccess
∗.txt	5	kampret.txt, data.txt, cpaccount.txt
/proc/∗	4	cpuinfo, meminfo, partitions, version

makes intuitively sense since in the beginning different attackers discover the homephoning requests at different times and visit these shell-promising URLs. Since our honeypot server always responds with an HTTP 404 *Page not found* error, the attackers slowly start giving up on their potential targets, perhaps attributing these errors to a successful "cleanup" by the website administrator.

6. ANALYSIS OF SERVER-SIDE ACTIVITY

In the previous section we, among others, quantified the percentage of shells that perform server-side homephoning. Homephoning, however, is only one out of many possible operations that a server-residing web shell can perform. In this section, we provide a high-level overview of other shell-related server-side activity, focusing on the activity that results from merely loading a web shell's main page, i.e., activity not triggered by the shell-operating attacker.

To discover this server-side activity, we first automatically rewrite all 541 shells of our *DYN_ANALYSIS* set to interpose all functions of interest. We perform this interposing through the use of PHP's runkit framework [28] which allowed us to log the calling of arbitrary system functions, together with their arguments, before allowing these functions to proceed. In our case, the functions of interest include functions that allow a shell to execute system commands (e.g., `exec`, `system`, and `shell_exec`), and functions for the reading and writing of files (e.g., `fopen`, `file_get_contents`, and `file_put_contents`). The modified shells are then visited using the OpenWPM framework, using the same setup as the for the discovery of server-side homephoning.

From our set of 541 shells, 170 shells (31.4%) call at least one of our monitored functions, with 128 shells executing system commands and 59 interacting with the file system. Figure 4 shows the overall popularity of our monitored PHP functions among shells, indicating a preference of the `exec` and `fopen` functions over their alternatives.

Table 2 shows the files that are commonly accessed and modified. Next to local files, some shells use `fopen` to open and read remote URLs. Among others, we witnessed the

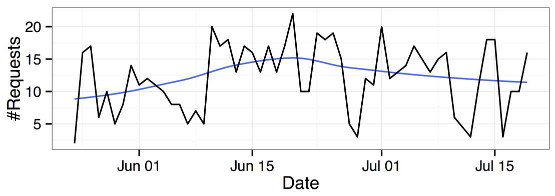

Figure 5: Daily number of attacker connections to our honeypot domains. The blue line shows a smoothed average of the same requests.

reading of remote configuration text files, the fetching of additional PHP code that expands the functionality of a shell, and the use of `fopen` as a method of server-side homephoning (e.g., `http://attacker.com/update.bin/admin.php?add=URL_of_the_shell`). In terms of local file accesses, some shells manifest their malicious behavior by reading and writing important system configuration files like `/etc/passwd` and `/etc/hosts`. The file `named.conf`, a configuration file of the BIND DNS server, is particularly popular among malicious web shells. Presumably, shells can modify BIND's configuration to change how a particular domain is resolved, or poison the local server with malicious DNS entries.

Table 3 presents groups of popular system calls. For the most part, shells collect different system information, including the current user and OS type, the file system, and even the list of running processes. An unexpected finding was that some shells check the system for the presence of specific applications. Besides queries for the version of Java and Perl, some shells use the `which` Linux utility to check for installed security-related software, such as, antivirus programs, (e.g., node32, and `drwebd`) and intrusion-detection systems (e.g. `snort,` and `rkhunter`). We also noticed that many shells check for Internet connection by using `wget` to download `http://www.google.com` and inspect the result. In some cases, instead of downloading the main page of `google.com`, the shells use `wget` to fetch a file ending with an image-related extension (e.g., `.jpg`), which they then rename to a `.pl` (Perl script) and execute it. We suspect shells follow this pattern in order to evade intrusion-detection systems looking for suspicious downloads.

7. ANALYSIS OF STALE DOMAINS

In Table 1, one can see that the most commonly contacted domain via client-side homephoning is the currently *parked* domain `alturks.com`. Parked domains are domains whose owners have given control of their domains to parking companies which populate them with advertisements and give a fraction of the advertisement profits to the original domain owner [8, 19]. Owners of parked domains often register domains that used to belong to individuals and companies but they were left to expire, either due to oversight, or because the original owner was no longer interested in the domain.

In the case of web shells and the `alturks.com` domain, one can reasonably assume that this domain used to belong to an individual or hacking group that had backdoored a large number of web shells in order to contact the `alturks.com` domain upon their loading. Even if that hacking group lost interest in the domain, the backdoored web shells circulating on the underground were never updated to remove the references to the included domain. Even though, in the case of `alturks.com`, the parking company is likely returning 404 error messages to the shells' requests for remote resources, this shows the possibility of discovering and even hijacking

Table 3: Commands that Shells silently execute

Group	Aggregated examples
Main System Info	id(71), uname(41), echo $OSTYPE(23), pwd(16), whoami(13)
More System Info	ls -la(7), df -h(2), uptime(2), ps aux(1), free -m(1)
Check Installed	which *(4), wget -help(5), javac -version(2), perl -v(1)
Internet Usage	cd /tmp;lynx|curl|GET *(8), wget|curl|lwp-download *(6)
Other actions	echo abcr57(19), killall -9 host(1), crontab (1)

Table 4: List of stale domains registered. Shaded entries are contacted via server-side homephoning.

Stale Domain	#Remote Requests	#Referring Domains
legal***.ru	4411	184
flyp****.us	1137	22
n**.org	749	23
nettekia****.com	521	33
evilc****.org	322	47
hack***.gen.tr	168	8
pira****.com	129	2
sil3nt****.com	24	1
Total	7461	311

new shell deployments by merely registering the appropriate domains which hacking groups allowed to expire.

This problem was first observed in the context of remote JavaScript inclusions by Nikiforakis et al. [24], who noticed that popular websites were occasionally referencing remote scripts of non-existing domains, a phenomenon which they called *stale domain-based inclusions*. By registering the appropriate domains, one could deliver malicious JavaScript code to popular websites and use it to steal session cookies and perform phishing attacks. In the case of web shells, a similar stale domain-based inclusion is significantly more harmful since web shells, by design, allow for server-side command execution. An attacker that can inject malicious JavaScript can effectively use it to send commands to the compromised server on which the shell is situated. Therefore, a stale domain-based inclusion in a web shell is sufficient not only to take control of the shell itself, but also to fully compromise the remote web server. Moreover, remote stale references for passive content, e.g., images, are sufficient to give away the location of new shells to anyone who cares to register the right domains.

By analyzing the homephoning requests of web shells, we discovered 11 domains which web shells were requesting resources from and, at the same time, were available for registration. Table 4 shows the eight domains that we registered, resolved to a host that we controlled, and returned HTTP 404 errors to all requests asking for content. We kept logs of these requests using the standard Apache web server logs. To avoid overestimating the problem due to visits by web crawlers, we only count the requests towards specific web-shell-related resources, e.g., `GET /yazciz/ciz.js HTTP/1.1`, and further filter out all requests whose reverse DNS resolution indicates that the IP address of the client that requests these resources belongs to a crawler.

Due to the different timelines involved with different top-level domain registrations, such as `.gen.tr` and `.ru`, not all domains became active at the same time. At the time of this writing, we have logs for three months of requests. Over this time period, we received a total of 7,461 requests for remote resources, belonging to 311 unique remote victim hosts. For requests associated with client-side homephoning, the `Referer` header of the attacker's browser leaked to us both the IP address of the shell-utilizing attacker, as well as the URLs of their shells and thus, implicitly, the domains of newly compromised servers. For server-side homephoning, where the compromised server directly connects to our server, we conservatively count every IP address as belonging to one website, even though, in cases of shared hosting, a web server could be hosting hundreds of different websites.

For ethical reasons, we never attempted to validate any of the shell installations claimed in the `Referer` headers of the requests in our logs. At the same time, these headers typically contained sufficient information to convince us that the requests are, in fact, originating from compromised servers. One of the most often reoccurring patterns was a `Referer` header exposing the directory structure of the compromised website, such as, `http://vict.im/admin/domainfonder.php?act=ls&d=/home/victim/public_html/&sort=0a`.

We recorded compromised websites all over the world, logging many instances of shells on websites of local businesses like an order service in Iran, a travel portal in China, a software company in Russia, a shipping company in US, and a college website in Morocco. A large fraction of the compromised websites had Iranian or Vietnamese TLDs. We also found cases that might have more severe consequences, e.g., a hospital website in Peru, a legal consultation office in Russia, a security services company in Vietnam. Our findings demonstrate that websites hosting such critical information are not exempt from the threat of malicious shells. We informed the victim websites about the signs of intrusion.

Another interesting observation is that, for most of our registered stale domains, we observed requests where the `Referer` header was either from “localhost” or from “127.0.0.1” with various high-number ports and paths. Since this header can only be emitted if a web shell and the attacker’s browser is situated on the same local machine, we can safely deduce that these requests are because attackers and researchers are trying out shells on web servers on their own machines. This means that an attacker who has control of the appropriate stale domains can now fully compromise machines that would otherwise be inaccessible via the public Internet.

The silver lining of our findings is that security companies can start monitoring domains involved with homephoning, anticipate their expiration, and register them before an attacker, or an unsuspecting domain parker, does. This will allow security companies to either automatically disarm these shells or, at the very least, provide them with a highly accurate method of identifying newly compromised hosts.

8. RELATED WORK

Even though web shells have been discussed in previous research, they are either only mentioned in passing, or treated as malicious ”black boxes” that need to be detected [23, 30, 32–35, 37], rather than understood.

Canali and Balzarotti, in their large-scale web honeypot experiment, deployed 17 publicly accessible web shells that attackers could discover through the use of specially crafted search engine queries, also known as *dorks* [11]. In addition to luring attackers to existing shells and studying their behavior, the authors noticed that if attackers discover a vulnerability that allows them to upload files, in 46% of the cases, they will upload a web shell and use that shell to interact with the compromised server. Interestingly, the authors observed that attackers who would discover already deployed shells, would use them to upload their own custom shells and then switch to their own shells for further interactions with the compromised host. The authors theorized that this is because attackers know that the existing shell may have a backdoor that would alert the original creator of their presence. As we showed earlier in our study, many web shells are indeed backdoored but these backdoors are triggered as soon as the main page of the shell loads, making the aforementioned evasive behavior completely futile.

In a follow up study that investigated the role of web hosting providers in detecting compromised websites, Canali et al. [12], among others, uploaded the c99 shell (arguably the most popular web shell) to 22 shared hosting providers and used simulated attackers to send commands towards the server. Only one of the 22 investigated shared hosting providers identified the malicious shell, even among the providers offering security services at an additional cost.

Kim et al. [20] compiled a collection of shells for benchmarking the detection rate of popular shell-detecting tools. The authors discovered that the tools either detected only well known shells and ignored less known ones, or marked a large number of benign files as “suspicious”, offloading the work of verifying a script’s maliciousness to a human analyst. Since the authors did not attempt any normalization or deobfuscation of their collected shells, it is possible that their compiled dataset contained multiple obfuscated versions of the same basic shells, which could be biasing their results against the tested web-shell-detecting tools. For this reason, we opted to compile our own set of malicious web shells and follow the normalization routines described in Section 3.

Finally, it is worthwhile noting that web shells are not the first hacking tools that contain backdoors. Cova et al. [13] analyzed freely available phishing kits and discovered that whenever a victim types in her credentials in a phishing page generated by a phishing kit, the credentials are not only made available to the user of the phishing kit, but also to various third parties. In another instance of backdoored hacking tools, Anti-CNN, a DDoS tool that was specifically targeting `cnn.com`, was also backdooring the machines of users that were participating in the DDoS attack [22].

9. CONCLUSION

In this paper, we presented the first comprehensive analysis of malicious web shells, by compiling a set of hundreds of malicious PHP shells and using a combination of static and dynamic analysis techniques to uncover the shells’ visible and invisible features. We showed that modern shells provide a wide range of tools to an attacker, ranging from remote command execution and portscanning, to password brute-forcing and privilege escalation. In addition, we provide evidence that a substantial fraction of the analyzed shells contain hidden backdoors and homephoning functionality, which attackers abuse to obtain access to compromised servers. We studied the attackers visiting their homephoning shells using honeypots, and described how other attackers, as well as security companies, can use expired domains to completely take-over malicious shells.

We argue that a better understanding of malicious web shells will naturally result into designing better detection techniques. Therefore, it is our hope that this paper and the corresponding datasets that we will be making available to other researchers, can be used to foster new research in the area of web application malware.

Acknowledgments: This work was supported by the Office of Naval Research (ONR) under grant N00014-16-1-2264 and by the National Science Foundation (NSF) under grant CNS-1527086.

10. REFERENCES

[1] JohnTroony's php-webshells repository. https://github.com/JohnTroony/php-webshells.

[2] Nikicat's web-malware-collection repository. https://github.com/nikicat/web-malware-collection/tree/master/Backdoors/PHP.

[3] Tennc's webshell repository. https://github.com/tennc/webshell/.

[4] UnPHP, the Online PHP Decoder. http://www.unphp.net/.

[5] Web Shells and RFIs Collection. http://www.irongeek.com/i.php?page=webshells-and-rfis.

[6] M. Abu Rajab, J. Zarfoss, F. Monrose, and A. Terzis. A Multifaceted Approach to Understanding the Botnet Phenomenon. In *Internet Measurement Conference (IMC)*, 2006.

[7] A. Aiken et al. Moss: A system for detecting software plagiarism, 2005. https://theory.stanford.edu/~aiken/moss/.

[8] S. Alrwais, K. Yuan, E. Alowaisheq, Z. Li, and X. Wang. Understanding the Dark Side of Domain Parking. In *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[9] VirusTotal. https://www.virustotal.com/.

[10] A. Brandt. Malicious PHP Scripts on the Rise. http://www.webroot.com/blog/2011/02/22/malicious-php-scripts-on-the-rise/.

[11] D. Canali and D. Balzarotti. Behind the Scenes of Online Attacks: an Analysis of Exploitation Behaviors on the Web. In *20th Annual Network & Distributed System Security Symposium (NDSS)*, 2013.

[12] D. Canali, D. Balzarotti, and A. Francillon. The role of web hosting providers in detecting compromised websites. In *Proceedings of the 22Nd International Conference on World Wide Web*, pages 177–188, 2013.

[13] M. Cova, C. Kruegel, and G. Vigna. There is no free phish: An analysis of "free" and live phishing kits. In *Proceedings of the 2Nd Conference on USENIX Workshop on Offensive Technologies (WOOT)*, pages 4:1–4:8, 2008.

[14] J. Dahse and T. Holz. Simulation of Built-in PHP Features for Precise Static Code Analysis. In *Symposium on Network and Distributed System Security (NDSS)*, 2014.

[15] Google Hacking Database (GHDB). https://www.exploit-db.com/google-hacking-database/.

[16] S. Englehardt, C. Eubank, P. Zimmerman, D. Reisman, and A. Narayanan. OpenWPM: An Automated Platform for Web Privacy Measurement. Manuscript, 2015.

[17] C. Holmes. Malware Lateral Movement: A Primer. https://www.fireeye.com/blog/executive-perspective/2015/08/malware_lateral_move.html, 2015.

[18] T. Holz. A Short Visit to the Bot Zoo. *Security & Privacy, IEEE*, 3(3):76–79, 2005.

[19] D. Kesmodel. *The Domain Game: How People Get Rich from Internet Domain Names*. Xlibris Corporation, 2008.

[20] J. Kim, D.-H. Yoo, H. Jang, and K. Jeong. "webshark 1.0: A benchmark collection for malicious web shell detection. In *Journal of Information Processing Systems (JIPS)*, 2015.

[21] J. Kornblum. Identifying Almost Identical Files Using Context Triggered Piecewise Hashing. *Digit. Investig.*, 3, 2006.

[22] J. Nazario. http://web.archive.org/web/20120722073150/http://ddos.arbornetworks.com/2008/04/netbot-attacker-anti-cnn-tool/, 2008.

[23] NeoPI: Detection of web shells using statistical methods. https://github.com/Neohapsis/NeoPI.

[24] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. You are what you include: Large-scale evaluation of remote javascript inclusions. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 736–747. ACM, 2012.

[25] OWASP : Testing for Local File Inclusion. https://www.owasp.org/index.php/Testing_for_Local_File_Inclusion.

[26] OWASP : Testing for Remote File Inclusion. https://www.owasp.org/index.php/Testing_for_Remote_File_Inclusion.

[27] OWASP : Unrestricted File Upload. https://www.owasp.org/index.php/Unrestricted_File_Upload.

[28] PHP: runkit Functions - Manual. http://php.net/manual/en/ref.runkit.php.

[29] PHP: Using Register Globals - Manual. http://php.net/manual/en/security.globals.php.

[30] R-fx Networks. Linux Malware Detect. https://www.rfxn.com/projects/linux-malware-detect/.

[31] R57 Shell | C99 Shell | Shell | TXT Shell | R57.php | c99.php | r57shell.net. http://www.r57shell.net/.

[32] Web Shell Detector. http://www.shelldetector.com/.

[33] Webserver Malware Scanner. http://sourceforge.net/projects/smscanner/.

[34] PHP Shell Detector – web shell detection tool. http://www.emposha.com/security/php-shell-detector-web-shell-detection-tool.html, 2011.

[35] M. Stowe. PHP Malicious Code Scanner. http://www.mikestowe.com/2010/10/php-malicious-code-scanner.php.

[36] H. Sverre. c99.php - phpshell. https://helgesverre.com/c99.php.

[37] T. D. Tu, C. Guang, G. Xiaojun, and P. Wubin. Webshell detection techniques in web applications. In *Computing, Communication and Networking Technologies (ICCCNT), 2014 International Conference on*, pages 1–7. IEEE, 2014.

APPENDIX

A. EXCURSUS ON IRC BOTS

During our manual code analysis, we found 18 web shells that implement an IRC bot in PHP in order to connect to an IRC server and become part of a bot network (i.e., a network of compromised machines under the control of an attacker [6, 18]). While most IRC bots try to connect to an unavailable IRC server (i.e., a stale domain) or to an empty IRC channel (i.e., the server is configured to not reveal any information), we found a case of an active bot network that is closely related to the analysis results presented in this paper. In the following, we provide a brief overview of our findings.

The referred IRC server had an uptime of four month and the geolocation of the IP address suggests that the server is located in Ukraine. However, certain names and the language of choice used in the IRC channel suggest that the attackers have an Indonesian origin. The analyzed PHP bot connects to a *#DdOs* channel with the nickname `[M][crew]123`. Based on the source code, we learned that *M* indicates that the compromised server runs an *Apache* web server and *123* is a random three digit number. In the *#DdOs* channel, we could identify 20 other bots following this name format, indicating compromised web servers running the same PHP bot. Additionally, other nicknames following different naming formats were present that likely belong to other bots. Based on these names, we estimate a bot network size of around 100 active bots during any given time of the day.

Once connected, the bot sends its hosted OS information to a private IRC channel and waits for commands that can be provided via private IRC messages. The bot's source code shows that after a successful login with a hardcoded password, the botmaster can force the bot to reveal the compromised system's information, send spam emails, execute system commands, download files, scan ports, as well as initiate TCP and UDP flooding attacks. The authentication can also be limited to a fixed IP address. We extended our copy of the PHP IRC bot with logging capabilities and removed potentially malicious code (to prevent an abuse from our system), and connected to the IRC server. However, our honey-bot did not receive any commands in a period of 48 hours.

Furthermore, we discovered another interesting IRC channel on the same IRC server. Here, URLs of vulnerable websites are reported by two crawler bots that seem to be hosted in Russia and the Netherlands. For example, several URLs referenced outdated installations of the PHP e-commerce software *osCommerce*. Based on the request path, it is evident that a *remote file upload* vulnerability is targeted that was disclosed in 2010. The nicknames of the bots suggest that the vulnerable websites were identified by using a variety of search engines, such as *Google*, *Bing*, *Yandex*, *Seznam*, *Walla*, *Biglobe*, and *Ask*. Moreover, the vulnerable websites are exploited automatically: we could observe that about six seconds after a vulnerable URL was posted to the channel, a new PHP bot following the aforementioned nickname schema connected to the *#DdOs* channel. Its IP address could be resolved to the very same domain in the previously reported vulnerable URL in the second channel, indicating a successful exploitation and infection with the PHP bot code.