Towards an Automatic Generation of Low-Interaction Web Application Honeypots

Marius Musch
TU Braunschweig
m.musch@tu-bs.de

Martin Härterich
SAP Security Research
martin.haerterich@sap.com

Martin Johns
TU Braunschweig
m.johns@tu-bs.de

ABSTRACT

Low-interaction honeypots (LIHPs) are a well-established tool to monitor malicious activities by emulating the appearance and behavior of a real system. However, existing honeypots share a common problem: Anyone aware of their existence can easily fingerprint and subsequently avoid them.

In this paper, we present Chameleon, our work towards an automatic generation of LIHPs for web applications. Chameleon creates honeypot versions of existing systems through automatic network interaction with the real application and builds response templates from the observed response traffic. By comparing similar responses, variable parts are identified and imitated with these templates. On run-time, the best matching template is chosen to respond to an incoming network request. This approach allows a large-scale deployment of Honeypots in a highly scalable fashion: No manual effort is needed in honeypot generation and a single instance of Chameleon can emulate a large number of heterogeneous systems simultaneously. Thus, a LIHP infrastructure for a company's full application landscape can be created, deployed and operated automatically with little effort and minimal technical resource requirements in a timely fashion.

We document our prototypical implementation for HTTP(S) and our practical experiments with the generated honeypots in the wild. The results are promising: The generated honeypots are indistinguishable for popular fingerprinting tools and the received traffic shows no difference to traffic directed at real systems.

1 INTRODUCTION

Our networked world is growing steadily and fast. Nowadays, networked applications are ubiquitous, ranging from small, single purpose IoT devices, over outsourced cloud services, up to highly complex business application landscapes. And all these systems have one thing in common: They offer public interfaces for potentially untrusted parties to interact with, often in the form of HTTP(S) servers.

Unfortunately, many of these systems are insecure, either due to insecure configuration, such as publicly known and unchanged default passwords, or because of security vulnerabilities, both disclosed and zero-days. This poses a significant challenge for security professionals: It is often unknown which class of systems are currently targeted by malicious parties, until it is too late and the community is surprised with large scale attacks, such as the mass-compromise of web cams and DVRs in 2016 [4].

ARES 2018, Hamburg, Germany

978-1-4503-6448-5/18/08…$15.00
DOI: 10.1145/3230833.3230839

1.1 Problem Statement

Honeypots are a well-established tool to monitor malicious activities and to detect previously unknown attacks. However, operating a real and intentionally vulnerable system, a so-called *high interaction honeypot* (HIHP), is both risky and time-consuming [7, 14] and thus *low-interaction honeypots* (LIHP) are often used instead. A LIHP is a dedicated networked application that emulates the appearance and behavior of a real system by providing the same public interfaces and exposing similar behavior, with the goal to observe unsolicited malicious traffic.

Still, even the set-up of a realistic LIHP is a non-trivial task, especially in respect of concealing the honeypot nature of the tool. LIHPs are particularly vulnerable to fingerprinting, as the behavior of the emulation might differ from the real implementation. The majority of web application honeypots, like HIHAT [13], DShield [12] and GHH [5], only serve static websites, which enables efficient honeypot fingerprinting by the attackers. Even more sophisticated honeypots like Glastopf [15] can be detected by fingerprinting techniques, as shown by Sysman et al. [17].

Similar to a trap without camouflage, attackers can simply avoid any honeypot they are aware of. Therefore, additional work to create new, unique honeypot instances for each emulated system is required [8]. Furthermore, to get a comprehensive insight in the current attack landscape, it is necessary to emulate a large range of systems and applications. However, creation and operation of individual, non-trivial LIHPs is costly both in respect of required manual effort and computational resources.

1.2 Chameleon

To address the outlined problem, we present Chameleon – an approach to automatically create LIHPs, emulating the pre-authentication surface of any given web server. Through comprehensive examination of the original system's services and behavior, Chameleon "learns" how the target systems reacts to various requests, especially in respect of static and dynamic ranges in the communication. While the automated imitation of a program by pure observation without knowledge of its inner workings is probably an impossible task, we do not have to solve this problem in general. Our main goal is merely to send plausible responses to requests sent by attack tools. This simplifies the problem, as exploit tools usually only use a limited subset of all possible interactions [3].

Chameleon's automatic generation allows for the quick creation of LIHPs of *many different* web applications, or of *many different*

versions of the same application. While one specific instance of the honeypot might still get fingerprinted, identifying them all without also labelling real applications as honeypots becomes increasingly difficult. Thus, the approach of automatically generating honeypots solves the problem of manual work and lowers the chances of successful fingerprinting while also vastly increasing the scalability of large web LIHP deployments.

2 CONCEPT

In this section, we provide a comprehensive overview on our approach towards honeypot generation and operation. Please note, that Chameleon's current objective is to create low-interaction honeypots for the *pre-authentication* surface of the emulated systems. We deem this attack surface to be sufficient, as we currently want to know *which* systems are targeted and *how* attackers would try to gain access – not what they do on a compromised system afterwards.

2.1 Design Goals

First, we briefly document the goals that steered the design:

Scalability. Chameleon is targeted to be able to create and operate large number of individual low-interaction honeypots, each emulating an individual, specific system. Thus, both the creation and the operation of honeypots have to be possible in a scalable fashion.

Automation. No manual intervention should be required to create and deploy a given honeypot. The overall goal of Chameleon is to be able to convincingly simulate a multitude of applications in various versions. Manual effort in honeypot creation would severely limit the approach's overall scalability.

Universality. The honeypot should be able to emulate any existing web application, regardless of the technology stack used by the original. This ensures creation of a wide variety of honeypots without being limited to specific technologies (like apps written in PHP as in [9]).

Deception. The generated honeypot should approximate indistinguishability from the real system. This goal especially applies to deceiving automated observers, such as fingerprinting, vulnerability testing, or mass-scanning tools.

2.2 High-level Overview

Chameleon creates honeypot versions of given web applications in a fully automatic fashion. In order to do so, Chameleon first repeatedly accesses the public HTTP interface with a crawling component, to create dynamic *response templates*. In operation mode, Chameleon uses these templates to convincingly emulate the original web application. As no real application logic is implemented and Chameleon only "parrots" learned responses back, the operation of a given system requires only very little computational resources. This allows to host a large number of individual honeypots at the same time.

2.3 Architecture

Generating a honeypot can be broken down in three major phases, each of which depends on the results of the previous phase (cf. Figure 1). Both the probing and the parsing phase is only executed once during the creation of each individual honeypot, while the publishing on the other hand is a continuous process, always running on the server hosting the honeypot.

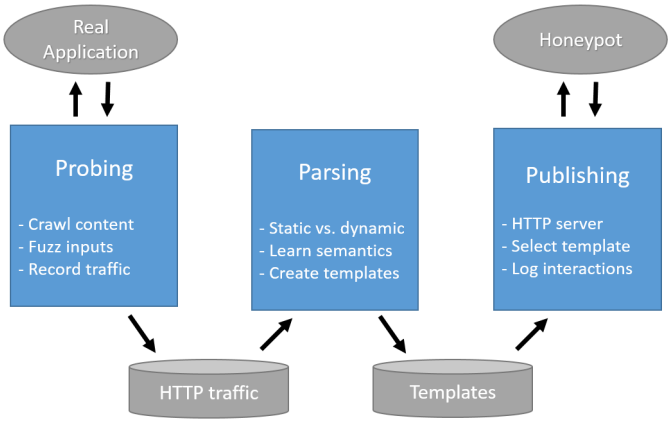

Figure 1: The Chameleon. Ovals are applications, rectangles represent phases and the disks are databases.

Probing: Starting with one or multiple URLs as input, the first step is to probe the web application by interacting with it and analyzing the responses. By extracting all hyperlinks in the responses, the automated crawler can recursively traverse and map the application. All these request–response pairs are stored for later processing. Since the responses observed during this phase are the sole base on which we later build the honeypot, it is important not only to find as many resources on the server as possible, but also to identify the range of different responses. For example, a login page might return multiple different error messages depending on the submitted input. To achieve a thorough, automatic observation, we designed and integrated a fuzzer into our web crawler, which mutates inputs like HTTP headers and URL query parameters (more on this in Sec. 3).

Parsing: The next step is to process all these stored request–response pairs from the previous phase. Here the most valuable information for the parser is not the actual content of the collected responses, but the variances in the responses caused by identical or similar requests. If the server always sends the same response for a particular resource, even when fuzzing slightly mutates the request, then it is safe to assume that this resource is static. For all non-static resources, Chameleon utilizes comparative analysis on multiple responses to similar requests, in order to identify the dynamic components of the resource and their respective value range. The result of this phase are templates containing the static code of the page like HTML, JavaScript, etc. but with the dynamic parts replaced using placeholders in Chameleon's custom syntax. If similar enough, several responses will be consolidated into one single template.

Publishing: After honeypot deployment, Chameleon utilizes the generated templates to respond to HTTP requests from attackers. The main challenge here is to find the best response given an arbitrary, new request that might not have been observed before. Therefore, a metric to compare HTTP requests and score their similarity was designed. Furthermore, a flexible underlying HTTP

server is required so that many different other servers can be imitated. All interactions with the honeypot are logged to serve as a basis for evaluations on attack trends.

2.4 Details on Parsing

Diff Creation. As the prober requested each resource on the server multiple times and with different inputs, the parser can now use this gathered information to find dynamic parts in the responses. Therefore, a meaningful representation of the differences between multiple similiar responses is required. Such a diff consists of a sequence of insertions and deletions that are applied to the first string to transform it into the second [6]. To address for minified files, a byte-wise diff is used instead of the usual line-based approach.

Variables. Based on the list of differences, the parser can start interpreting them. From here on, a deletion directly followed by an insertion of roughly equal length will be refered to as a substitution. Each substitution is analzyed and categorized into one of the following types:

Random Tokens are arbitrary strings or numbers with no easily observable pattern to them, e.g. cryptographic nonces. They can be found anywhere in the headers, body or even URL.

Session Tokens are a special case of random tokens, bound to a session and with a longer lifetime than random tokens. Session tokens are most often found in cookies and or a URL.

Timestamps are some representation of the time, be it the time of the day, the date or both together. There are many different possible ways to display the same timestamp, for example `09.01.2018` and `Mon, 9 Jan 2018 09:39:48`.

Reflections occur when input is copied into the output, which is often the case in HTML forms. For example, if a search form is submitted, many websites will show the search term again on the results page.

Unknown is used to describe any variable that fits into none of the above categories. These are usually changes in natural text on the rendered page or in the code behind that page. Inferring the semantics of such changes would infeasibly complex and thus they are ignored.

Variable Detection. Timestamps cannot be detected purely on the basis of the diff, as they might only rarely change. For example, if the page prints out the current date, but not the current time. Therefore, timestamps are detected via various regular expressions, as neither the current local time of the server is known, nor the format to used to represent that time. All the remaining types of variables are solely detected based on the diff. To find reflections, the parser looks at each of the substitutions in the request and searches the response for occurrences of exactly the same substitution. The remaining substitutions are either of type random token, session token or unknown. To make sure that changes in natural text or code are not erroneously identified as tokens, there is an additional requirement to the length of the change. To differentiate between random and session tokens requires that the prober initially made multiple requests with different cookies. Substitions occurring only when the cookie changed have to be session tokens, while substitutions changing regardless of the cookies are random tokens. Should the variable be neither of all these types, is considered of type unknown and ignored.

Template Generation. The last step of the parsing is to create templates of the request-response pairs collected during the probing. In these templates the dynamic parts are described by placeholders based on the variable detection from the previous step.

As the prober requested each page multiple times and also with fuzzed inputs, the parser now has to decide for each of these responses if it can be merged into an existing template. For example, the same login page might return different error messages, depending on the input. If the template contains a substitution of type unknown or if there is one long isolated deletion or insertion, that template is considered to be not mergeable with other templates and saved separately.

2.5 Details on Publishing

During operation mode, Chameleon uses the previously created templates to respond to new requests. Given an arbitrary HTTP request, the first step is to find the best template to use for the response. Ideally the honeypot would always return exactly what the attacker expects, however in practice there will be cases where requests are ambiguous and multiple templates could be used, or cases where there might be no template at all matching the requested resource.

As a basis for a similarity metric for HTTP requests, the different parts of a request were ordered by their significance for the response. Starting with the most important, the following ranking was created: HTTP method, path of the URL, HTTP body (if PUT/POST) or query of the URL (otherwise), HTTP headers. The first two, the method and path, are deemed so important that they have to match exactly. If there is exactly one template where these both match, we use that template. On the other hand, if there is no template satisfying this requirement, the publisher searches all templates for responses with error code `404` and selects one of these. This indicates that the requested resource is not available in the same manner the real application would present such an error.

If there is more than one template with matching method and path, the publisher has to decide which of these to take. For that, it is assumed the best response is the one obtained with the most similar observed request. Therefore, we compare the body, query part and headers of each stored request with the the newly recieved one, calculate the similarity of those key-value pairs and choose the template corresponding to the most similar request obtained during the probing.

After selecting the most fitting template, a HTTP response is generated from that template, which requires iterating over all variables in the template and replacing them with actual content:

Random Tokens are newly generated, with constraints to preserve the characteristics of the observed values. Uppercase and lowercase letters, as well as numbers are randomly changed within their respective sets, while all other characters are preserved. This has the effect of generating new tokens with similar statistical distributions as the observed tokens.

Session Tokens are generated in the same manner as the random tokens described above. If the same cookies are received again some time later, instead of generating new values for the session

tokens, the stored values are used and only new values for the random tokens are generated.

Timestamps are just replaced by the current time, formatted in the same manner as observed during the probing. However, timestamps with specific semantics, such as the `Expires` or the `Last-Modified` headers, are not replaced, as such unusual behavior could reveal the honeypot.

Reflections are not possible to determine by only looking at the template, as they are dependent on the current request. Thus the publisher has to find the position of the reflected part of the input in the newly received request and extract the value from there.

3 IMPLEMENTATION

The prober uses *HTMLUnit*[1] *v2.22*, a GUI-less browser written in Java that simulates *Chrome v51*. During the recursive crawl of the web application all resources referenced in `href` and `src` attributes of HTML pages are extracted. The prober follows redirects, executes JavaScript, records all XMLHttpRequests and submits HTML forms to discover as many resources as possible on the remote server. Common files like `robots.txt` and `sitemap.xml` are also requested to discover new URLs, which are possibly not linked in other locations. Furthermore, requests are fuzzed by changing the HTTP method and mutating the headers (esp. the cookies) and the URL query to learn more about the behavior of the application with respect to changes in the requests. The fuzzer uses a mutation based approach and replaces the values with similar character sequences of the same length.

The parser first preprocesses all responses with a variety of regular expressions, to find dynamic values like date fields that rarely change. Then the diff is applied to the responses of similar requests, using *Myers' diff algorithm* [10] followed by a semantic cleanup to remove short, coincidental equalities within longer changes. This cleanup removes short equalities within longer changes as can be found even inside completely random hexadecimal SIDs [2].

The publisher is based on *NanoHTTPD*[2] *v2.3.1*, a flexible, lightweight HTTP server implementation written in Java. The publisher not only imitates the HTTP body, but also all other parts of the response, especially including the HTTP headers. This way, Chameleon fully imitates the original HTTP server as closely as possible. Depending on the copied, original system, Chameleon can appear to be, e.g., an Apache HTTPd or a Microsoft Internet Information server, without any configuration effort for the honeypot operator. The current proof-of-concept implementation of Chameleon's publishing unit does not yet support all HTTP request headers that potentially alter the server response. For instance, fully supporting the `Content-Encoding` header would require an implementation of all possible compression algorithms to fully emulate the original web server. We leave this implementation task to a future, productive iteration of Chameleon.

4 PRACTICAL EVALUATION

While the process of generating new honeypots from existing applications is highly automated, their evaluation is not. Furthermore, running real applications to compare them to the honeypots requires a lot of resources and thus limits the number of applications that can be evaluated. This means only a handful of applications are used in this evaluation, despite the fact that the generation approach would easily allow for a far greater number of honeypots.

For the evaluation we selected the five most popular Content Management Systems (CMS) as they are widely used on the Internet. As time of writing and according to `w3techs.com`[3], these were WordPress, Joomla, Drupal, Magento and TYPO3. Blogger was excluded, as it cannot be self-hosted – a requirement for our evaluation to obtain the traffic logs, but not a limition of Chameleon in general. Using a fresh installation in its default state for each CMS, Chameleon automatically generated a honeypot for each of the five web applications.

4.1 Honeypot Creation

For the creation of the honeypots, we used one laptop with a 2.9 GHz dual-core CPU and 16GB of RAM. We started four probers simultaneously and crawled up to a link depth of four. On average, it took about 18 minutes to automatically prepare each honeypot using the probing and parsing. Roughly 256 unique templates per application were created, each on average containing 1.9 variables. Table 1 shows the absolute numbers per application.

Table 1: Generated response templates for the target CMSs

CMS	Version	Duration	Templates	Avg. Vars
Drupal	8.2.1	9 min	190	1.9
Joomla	3.6.3	12 min	139	2.7
Magento	2.1.2	38 min	451	1.3
TYPO3	7.6.14	22 min	414	2.2
WordPress	4.6.1	8 min	85	2.1

Further analysis of the generated templates showed that variables of type timestamp are the most common, as they are present in every `Date` response header. Random tokens and reflections are concentrated on a small number of templates, as a lot of the resources are static content like images and stylesheets. Only a few session tokens were found, but this is to be expected, as only pages accessible before authentication were crawled. For more details for each type of variable, see table 2.

Table 2: Minimum, maximum and average number of variables per template for each type of variable

Type of Variable	Minimum	Maximum	Average
Random Tokens	0	45	0.65
Session Tokens	0	3	0.01
Reflections	0	18	0.19
Timestamps	1	4	1.05

[1] http://htmlunit.sourceforge.net/
[2] https://github.com/NanoHttpd/nanohttpd
[3] https://w3techs.com/technologies/overview/content_management/all

4.2 Practical Results

Fingerprinting. To evaluate the deception of our five generated honeypots, we used three open-source fingerprinting tools: *Nmap*[4], *WhatWeb*[5] and *lbmap*[6]. While nmap is mainly used for port scanning, its `-V` option tries to detect the version of known services with open ports. The other two tools are specifically designed for HTTP fingerprinting. While even the real application was sometimes not correctly identified, the tools always produced the same output for both the real CMS and the generated honeypot mimicking it. Therefore, deception was achieved as attackers cannot use currently available fingerprinting tools to identify our honeypots.

Empirical Study I. Next, we gathered empirical data by deploying both the real applications and the honeypots on the Internet for a period of four weeks. The Chameleon was running in Amazon's EC2 on a single *T2.micro* instance with one shared vCPU and 1GB of RAM. While our analysis of the requested URLs showed no significant difference between the two systems, we only received 1.504 requests from 206 different IP addresses, not counting duplicate requests from the same IP within one minute. The cause for this might be that our systems were deployed on newly registered domains not indexed by popular search machines.

Empirical Study II. To quickly increase the amount of gathered data, we replaced a production system, in this case a WordPress blog of one of the authors, with a generated honeypot. With the honeypot running the site, human visitors could no longer post comments, but otherwise the site was still functional and served all existing blog posts. We then compared two weeks of logs from the real system with two weeks of logs from the Chameleon. Overall, we received a total of 13,595 unique requests from 1,712 different IP addresses. With this amount of traffic for a single system, it makes more sense to compare the interactions on the real systems and the honeypot.

Table 3: Ten most common requested paths on the real site compared with number of requests on the honeypot. Nine of these were also the most requested paths on the honeypot. Filenames in italics were replaced to describe their function

Path	Real system	Honeypot
`wp-login.php`	932	887
`/`	613	707
`xml-rpc.php`	448	398
Blog post 1	183	828
`robots.txt`	182	151
Blog post 2	85	53
Blog post 3	73	51
Category 1	66	64
`style.css`	66	64
`wp-emoji-release.min.js`	58	50

Table 3 shows that the number of requests is roughly equal on both systems for most files. Only one of the blog posts attracted far more traffic on the honeypot than the real system. Investigating the template of this post revealed a bug in the parser, which caused some crawlers to run into loops on that particular page as the query in one self-referencing link constantly changed. Other than that, the similar behavior on both systems shows that the honeypot successfully deceives attackers.

Further investigation of the POST requests received by the honeypot revealed various attempts to take over the alleged WordPress. Many attackers tried to log in with common passwords like `skater`, `test1234` or `adminpass`. Previously, the correct username seems to have been obtained by extracting the author's name from one of the blog posts, which equals the username by default. About half of these attempts used the regular login at `wp-login.php`, while the other half tried to exploit a login amplification attack at `xmlrpc.php`. Most of the remaining attacks focused on exploiting vulnerable plugins or themes, which might have been installed by the administrator. Additionally, there were various attempts to upload malicious files to the server via `wp-admin/admin-ajax.php`. Table 4 lists the exact number of unique POST requests to security related paths.

Table 4: Attempted attacks on the honeypot

Attack type	Occurrences
Login via `xml-rpc.php`	401
Login via `wp-login.php`	389
Test for vulnerable plugin	128
Test for vulnerable theme	21
File upload	16

Note that the number of POST requests to `wp-login.php` shown in table 4 is only roughly half of the amount of requests listed in table 3 to that path. This is due to the fact, that the login site was usually first requested with a GET request and then submitted with a POST request, thus each login attempt is counted twice in table 3.

5 RELATED WORK

Existing web honeypots: Among the first honeypots to focus on web servers were `HoneyWeb` [11] from 2002 and the `Google Hack Honeypot` [5] from 2005. Since then the sophistication of honeypots increased a lot and, as time of writing, `Glastopf` [15] is considered the state-of-the-art in web application honeypots. It can emulate thousands of vulnerabilities and replies to an attack by using a response the attacker is probably expecting. However, the vulnerable web pages themselves are still static and susceptible to fingerprinting. Therefore, manual work would be required to modify the appearance of each individual Glastopf instance.

In general, all these existing web server honeypots try to entice as many attackers as possible by appearing to be vulnerable to a wide range of attacks. This increases the amount of collected data at the cost of realism and deception. Each instance of the Chameleon on the other hand simulates one specific version of a real server and thus achieves far better deception. Furthermore, these existing honeypots only serve static files and thus can easily be fingerprinted, while the Chameleon emulates the dynamic behavior of real applications.

[4] https://nmap.org/
[5] https://github.com/urbanadventurer/WhatWeb
[6] https://github.com/wireghoul/lbmap

Academic approaches: The first academic publication that proposed a generic way to automatically generate new honeypots based on existing services was published in 2005 by Leita et al. [3]. The underlying goal of the approach is to "learn" unknown network protocols through the creation of the state machines. Thus, they collected all traffic from a high-interaction honeypot and derived a finite state machine to represent those exchanges. While the approach seems to be well suited for attacks over stateful protocols, the usefulness for stateless protocols like HTTP remains unclear. With a similar goal but different approach, Cui et al. [1] created `RolePlayer` in 2006, a generic system able to mimic many application protocols. Amongst other protocols, they tested their approach with HTTP but only with a static resource on the server. A restriction of their work is that new communication attempts with RolePlayer must be consistent with the learned "script", while Chameleon can handle requests in arbitrary order.

In 2008, Small et al. [16] published a generation approach which focusses on generic web honeypots. The goal of their approach is to *emulate vulnerabilities*. For this, they use natural language processing and string alignment techniques to automatically generate dynamic responses. In contrast, Chameleon is built from the ground up to *emulate applications*, rendering both approaches to be fundamentally different. Finally, in general, these existing academic approaches suffer from insufficient deception capabilities and did not solve the problem of honeypots getting fingerprinted. Thus, none of these techniques have similar capabilities as Chameleon when it comes to seamlessly blending into existing application landscapes.

6 FUTURE WORK

In the context of this paper, we have shown Chameleon's capabilities to easily create decoy web applications. This is an exciting first steps towards several compelling future research directions:

Foremost, we will leverage Chameleon-honeypots for long-term studies. Low-traffic low-profile web applications, as such honeypots instances are by nature, require significant time before they are recognized as potential attack targets. Thus, to collect meaningful information, the time frame of an experiment has to be expected to last months if not years. We are currently in the preparation phase for such a set-up. This will allow us to monitor the evolution of the automized landscape, via cross-referencing attack traffic on unrelated Chameleon instances, as well as the potential discovery of new targeted attacks on specific web application types.

Furthermore, the task of automatic web LIHP creation still offers avenues of further exploration: At this point Chameleon is only able to create exact carbon copies of existing systems. While a single Chameleon evades fingerprinting attacks due to its precise imitation of the original application, the existence of several instances of the same application might cause suspicions on the adversary side. Thus, we will experiment in *decoy morphing*. i.e., the merging of data-content from related Chameleon instances to create unique honeypots.

Finally, we are interested in exploring to which extent Chameleon's underlying concept of automatic honeypot generation can be applied to other, non-HTTP protocols.

7 CONCLUSION

In this paper, we presented Chameleon – an approach for a fully automatic creation of web LIHPs. We used Chameleon to create honeypot versions of several popular CMSs. The results of the practical experiments provide evidence, that our system functions as planned: For automated tools, the created honeypots are indistinguishable from the real versions of the emulated systems. Furthermore, thanks to Chameleon's templating system, the generated honeypots provide interactivity with potential human visitors that closely mirrors the behavior of the regular system under legitimate usage, adding further to the honeypot's deception capabilities. And finally, comparing the web traffic received by the honeypots with the traffic to the real systems, no apparent differences could be observed. The actual generation of the honeypot instances is fully automatic and requires no human effort. A single instance of Chameleon was capable to run all our honeypots simultaneously.

Thus, Chameleon is a viable concept to create, deploy and operate large numbers of heterogeneous LIHPs on a large scale, which enables several exiting future research streams.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable feedback. This work was supported by the German Federal Ministry of Education and Research (BMBF) through the project "BDSec – Big Data Security" (grant 01IS14009E).

REFERENCES

[1] Weidong Cui, Vern Paxson, Nicholas Weaver, and Randy H Katz. Protocol-independent adaptive replay of application dialog. In *NDSS*, 2006.
[2] Neil Fraser. Diff strategies. [online], https://neil.fraser.name/writing/diff/, 2006.
[3] Corrado Leita, Ken Mermoud, and Marc Dacier. Scriptgen: an automated script generation tool for honeyd. In *ACM ACSAC*, pages 12–pp. IEEE, 2005.
[4] Eric Limer. How hackers wrecked the internet using dvrs and webcams. [online], http://www.popularmechanics.com/technology/infrastructure/a23504/mirai-botnet-internet-of-things-ddos-attack/, October 2016.
[5] Ryan McGeehan and Greg Smith. Google hack honeypot. [online], http://ghh.sourceforge.net, 2005.
[6] Webb Miller and Eugene W Myers. A file comparison program. *Software: Practice and Experience*, 15(11):1025–1040, 1985.
[7] Iyatiti Mokube and Michele Adams. Honeypots: concepts, approaches, and challenges. In *Proceedings of the 45th annual southeast regional conference*, pages 321–326. ACM, 2007.
[8] B Mphago, O Bagwasi, B Phofuetsile, and H Hlomani. Deception in dynamic web application honeypots: Case of glastopf. In *Proceedings of the International Conference on Security and Management (SAM)*, page 104, 2015.
[9] Michael Mueter, Felix Freiling, Thorsten Holz, and Jeanna Matthews. A generic toolkit for converting web applications into high-interaction honeypots. *University of Mannheim*, 280, 2008.
[10] Eugene W Myers. Ano (nd) difference algorithm and its variations. *Algorithmica*, 1(1-4):251–266, 1986.
[11] Fabien Pouget and Marc Dacier. White paper: Honeypot, honeynet: A comparative survey. Technical report, RR-03-082, Institut Eurecom, 2003.
[12] DShield Project. Dshield web honeypot project. [online], https://sites.google.com/site/webhoneypotsite, 2011.
[13] HiHAT Project. High-interaction honeypot analysis tool. [online], http://hihat.sourceforge.net, 2007.
[14] Niels Provos and Thorsten Holz. *Virtual honeypots: from botnet tracking to intrusion detection*. Pearson Education, 2007.
[15] Lukas Rist, Sven Vetsch, Marcel Kossin, and Michael Mauer. Know your tools: Glastopf-a dynamic, low-interaction web application honeypot. *The Honeynet Project*, 4, 2010.
[16] Sam Small, Joshua Mason, Fabian Monrose, Niels Provos, and Adam Stubblefield. To catch a predator: A natural language approach for eliciting malicious payloads. In *USENIX Security Symposium*, pages 171–184, 2008.
[17] Dean Sysman, Gadi Evron, and Itamar Sher. Breaking honeypots for fun and profit. [online], https://media.ccc.de/v/32c3-7277-breaking_honeypots_for_fun_and_profit, 2015.