# Toward a Moving Target Defense for Web Applications

*(Invited Paper)*

Marthony Taguinod, Adam Doupé, Ziming Zhao, and Gail-Joon Ahn
Arizona State University
{mtaguino, doupe, zmzhao, gahn}@asu.edu

***Abstract*—Web applications are a critical component of the security ecosystem as they are often the "front door" for many companies; as such, vulnerabilities in web applications allow hackers access to companies' private data, which contains consumers' private financial information. Web applications are, by their nature, available to everyone, at anytime, from anywhere, and this includes attackers. Therefore, attackers have the opportunity to perform reconnaissance at their leisure, acquiring information on the layout and technologies of the web application, before launching an attack. However, the defender must be prepared for all possible attacks and does not have the luxury of performing reconnaissance on the attacker.**

**The idea behind Moving Target Defense (MTD) is to reduce the information asymmetry between the attacker and defender, ultimately rendering the reconnaissance information misleading or useless. In this paper we take the first steps of applying MTD concepts to web applications in order to create effective defensive layers. We first analyze the web application stack to understand where and how MTD can be applied. The key issue here is that an MTD application must prevent or disrupt a vulnerability or exploit, while still providing identical functionality. Then, we discuss our implementation of two MTD approaches, which can mitigate several classes of web application vulnerabilities or exploits. We hope that our discussion will help guide future research in applying the MTD concepts to the web application stack.**

## I. Introduction

Web applications remain the most popular way for businesses to provide services over the Internet. With more web applications available, more sensitive business and user data is managed and processed by web applications. Consequently, vulnerabilities in web applications put both businesses and end-users' security and privacy at risk.

This is not an abstract risk, as the JPMorgan Chase breach in 2014 affected 76 million US households [1]. Bloomberg reported that the hackers "exploited an overlooked flaw in one of the bank's websites" [2]. Thus, web applications are the "front door" for many companies, and therefore their security is of paramount importance.

Many techniques and tools using static analysis (white-box) or dynamic analysis (black-box) approaches have been proposed and developed to discover the vulnerabilities of web applications [3]–[7], so that the vulnerabilities can be removed before attackers discover and exploit them. However, the efforts of discovering and fixing vulnerabilities are not enough to protect web applications for many reasons: (1) the increasing complexity of modern web applications brings inevitable risks that cannot be fully mitigated in the process of web application development and deployment [8], and (2) attackers can take their time, to understand the web application's functionality and technology stack, before launching an attack.

We believe that a defense-in-depth approach is best to securing web applications. Therefore, to complement the aforementioned vulnerability analysis techniques, we propose to use the ideas of Moving Target Defense (MTD) to create a novel and proactive approach that adds an additional layer of defense to web applications. At a high level, a moving target defense dynamically configures and shifts systems over time to increase the uncertainty and complexity for attackers to perform probing and attacking [9], [10]. While a system's availability is preserved to legitimate users, the system components are changed in unpredictable ways to the attackers. Therefore, the attacker's window of attack opportunities decrease and the costs of attack increase. Even if an attacker succeeds in finding a vulnerability at one point, the vulnerability could be unavailable as the result of shifting the underlying system, which makes the environment more resilient against attacks.

To best apply the MTD ideas to protect web applications, there are two high-level decisions: (1) choose what component to move in a web application, and (2) decide the optimal time and how often to move components. To assist in answering these questions, we dissect the modern web application architecture, both client and server, and their running environments to explore the possibilities of applying MTD at different layers. We hope our analysis provides insights into the trade-offs among the different places to apply MTD to web applications.

We also discuss our first steps in applying MTD techniques to protect web applications. The first technique changes the server-side language used in a web application by automatically translating server-side web application code to another language in order to prevent Code Injection exploits. The second technique shifts the database used in a web application by transforming the backend SQL database into different implementations that speak different dialects in order to prevent SQL Injection exploits.

The main contributions of this paper are the following:

- We discuss the possibilities of applying moving target defense to different layers of web applications.

- We propose two novel approaches to changing the implementation language of a web application and the database implementation while keeping the functionality.

## II. Background

In order to properly understand how to apply the ideas of moving target defense to web applications, we first describe web application, then the ideas behind moving target defense.

### A. Web Applications

As shown in Figure 1, a web application follows a distributed application structure, with components running on the server and the client.

The client first uses the communication channels (typically the protocol HTTP and its derivative protocols, such as HTTPS, SPDY, and HTTP/2) to issue a request to the server-side component.

The server side typically includes the following layers from top to bottom[1]:

- The server-side logic layer implements the application business logic using high-level programming languages, such as Java, PHP, or Python.
- The web server layer receives the HTTP request from then client, parses the HTTP request, and passes the request to the appropriate server-side program. Examples include Apache web server, Windows IIS, or Nginx.
- The data storage layer that stores the web application state and user data. Popular data storage systems are traditional SQL databases, which include MySQL, PostgreSQL, or MSSQL.
- The operating system layer that provides the running environment for the web server layer and database storage layer.
- The infrastructure layer that runs the operating systems. An infrastructure could be a physical machine or a virtualization platform which manages multiple virtual machines.

The client receives the HTTP response from the server-side code, and the job of the client is to convert the HTML contained in the HTTP response into a graphical interface for the user. The client includes:

- The client-side logic layer that is usually called the presentation layer. This is written in a combination of HTML, CSS, and JavaScript, with JavaScript providing a way for the server-side code to execute application logic on the client.
- The browser retrieves the presentation layer code from the server, interprets it, and presents it as a graphical interface to the user.
- The storage layer that the presentation layer code uses to store data. Available storage methods include `cookies`, `localStorage`, `IndexedDB`, and File APIs.
- The operating system layer, which the browser runs on.

If a layer in Figure 1 is compromised, its upper layers are not trustworthy. For instance, if the server's operating system is compromised, then the data storage, web server, and server-side logic are all compromised. Because the presentation layer is created by the server and sent across the communication channel, a compromise of the server or the communication channel compromises the presentation layer. Adversaries can attack a layer in Figure 1 through its interfaces exposed to the upper layers.

For example, in a heap spraying attack on the client browser layer [11], an attacker allocates malicious objects using JavaScript in the presentation layer to coerce the browser to spray objects in the heap, increasing the success rate of an exploit that jumps to a location within the heap. In this case, the attacker uses the presentation layer to exploit a vulnerability in the browser layer that leads to arbitrary code execution in the browser's address space. The injected arbitrary code can in turn exploit a vulnerability in the client operating system to escalate its privilege and further infect the client machine. Furthermore, the malicious JavaScript code might be delivered by an attacker exploiting a vulnerability in the server-side logic layer, using a reflected or stored cross-site scripting (XSS) vulnerability.

### B. Moving Target Defense

The basic idea of moving target defense (MTD) is to continually shift and change system configurations over time to increase complexity and cost for attackers. MTD does not remove vulnerabilities directly but limits the exposure of vulnerabilities, so opportunities for attack can be decreased. In this way, MTD acts as part of a defense-in-depth strategy. The effectiveness of an MTD approach depends on how many components are moved (what-to-move) and the frequency of movement (when-to-move).

The widely adopted address space layout randomization (ASLR) [12] in modern operating systems is an instance of MTD. Existing ASLR mechanisms randomly arrange the address space positions of key data areas (what-to-move) of a process when it is launched (when-to-move), including the base of the executable and the positions of the stack, heap, and libraries. In this way, even if attackers are able to exploit a memory corruption vulnerability in a binary (such as the classic buffer overflow), it is difficult for them to transfer control flow to their injected shellcode, as they cannot predict the memory layout of the process.

MTD mechanisms for programs can be categorized into two classes depending on if a program is running (*dynamic*) or not (*static*) at the time when moving happens. For example, existing ASLR approaches are static, because the

[1]Of course, modern web application stacks can become increasingly complex, with caches, external requests, or other services, however in this paper we restrict our discussion to this abstracted model.

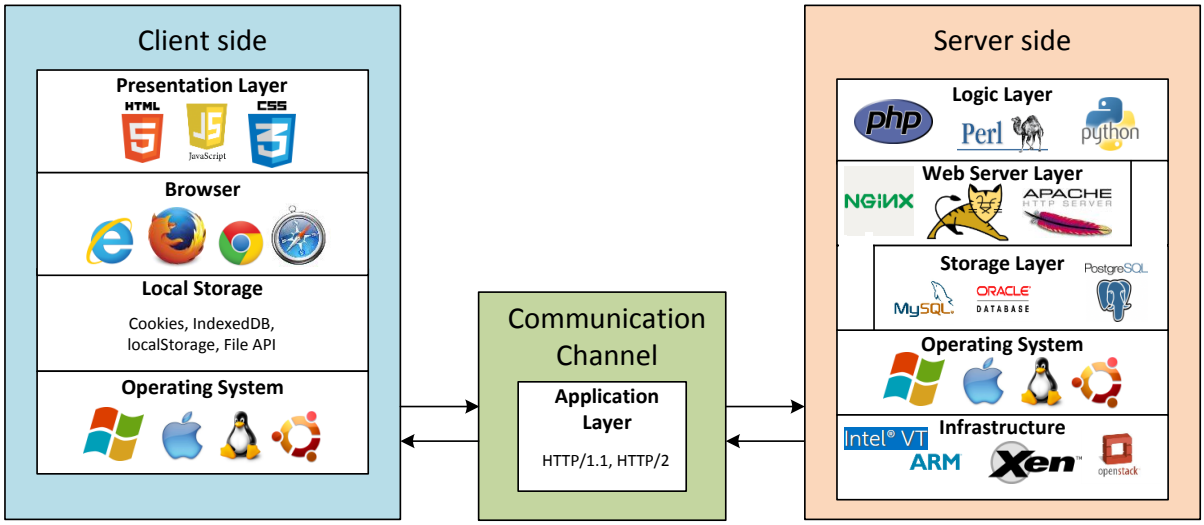


Figure 1. A Modern Web Application Architecture and Its Running Environments.

positions of code and data areas are only moved at the launch of a program but not when a program is running. A dynamic MTD offers more choices for when-to-move, but may be more difficult to implement.

## III. Moving Target Defense for Web Applications

The core idea of moving target defense (MTD) can be applied to every layer of web applications and their running environments. However, the key issue is that the "moving," when applied, must either prevent a vulnerability or exploit while at the same time not alter the application functionality. In this section, we discuss what components that are available for moving at the web application layers. We specifically focus on the layers specific to web applications: the logic layer, storage layer, and presentation layer. For other layers that are common to other applications, which include the operating system layer and the infrastructure layer, we refer the interested reader to research in these areas [12]–[21].

### A. Logic Layer

There are at least two ways to apply MTD at the logic layer by changing a web application's implementation. The first approach uses the idea of software diversity [14], which is widely used in lower-level languages, to change and modify the code at the statement, function, or object levels. This type of diversity is used to combat memory corruption vulnerabilities, specifically return-oriented programming exploits, which take advantage of previously known code-layouts. This automated diversity MTD technique can be done statically or dynamically. However, many web applications are written in higher-level languages, such as Java, Python, and Ruby, which are immune from memory corruption vulnerabilities. In fact, most web application vulnerabilities are inherent in the code itself, such as Cross-Site Scripting (XSS), where the server-side web application code creates HTML from unsanitized, untrusted attacker input. Software diversity does not handle this case, as the vulnerability is part of the web application's logic.

Another MTD approach is to switch a web application's implementation from one language to another, which could eliminate some language- or framework-specific vulnerabilities, as some vulnerability classes are specific to certain programming languages. For example, an application that is developed with *Ruby on Rails 3.0.5* may introduce execution-after-redirect vulnerabilities, while its counterpart developed with *Python* and *Django 1.2.5* is impervious to this class of vulnerabilities due to the different implementations of the underlying framework [22]. Changing the web application's implementation language could be either static or dynamic. In a static switch, the web server simply launches another language implementation of the application. To make the process automated, web application developers only need to develop the application once using the language they prefer, and a translator program translates their code into functionally equivalent code in the other language. The translation is difficult when the input language has some features that the output language can not offer. In a dynamic switch, the states of the running web application would need to be maintained or transformed for the program in

another language to understand. In Section IV we discuss our implementation of this idea.

### B. Storage Layer

The biggest challenge the storage layer faces are SQL injection attacks in which data from the logic layer is interpreted as SQL statements by the database management systems. In order to perform successful SQL injection attacks, attackers need to carefully craft their input by using some reversed tokens in the targeted SQL syntax in order to modify the logic layer developer's intended SQL statement.

While SQL itself is standard, different SQL database implementations use slightly different SQL syntaxes (also called dialects). Taking advantage of the fact that different databases use slightly different SQL syntaxes, switching the database used in a web application may defeat some SQL injection exploits that are targeted at a specific SQL dialect. For example, both single (`''`) and double (`""`) quotes are used for quoting values in MySQL—while PostgreSQL uses only single quotes for values, instead reserving double quotes for identifying field names, table names, etc.

Static MTD for the database can be realized by exporting the data from one database implementation and then importing it into a different database implementation. Dynamic MTD for storage layer can also be achieved if multiple, yet different, database instances are running and being continually synchronized. In Section V we discuss our implementation of this idea.

### C. Presentation Layer

The presentation layer contains technologies that are most directly accessible to the user. For instance, client-side JavaScript code running some of the web application's logic, the HTML DOM containing form information, and CSS that enables modification of the web layout. The most direct threat of the presentation layer are Cross-Site Scripting (XSS) attacks, where malicious scripts are injected into the web application in order to steal information from users.

There are techniques related to MTD that have been proposed to prevent against such attacks. One such technique is to introduce a degree of randomness to the underlying HTML form fields by adding tags to each field that hides their real values [18]. Another approach, targeting a different technology, is to introduce randomness to the JavaScript code by mutating tokens in such a way that the attacker cannot guess the correct token to inject in addition to running multiple versions of the website that each utilizes varying JavaScript versions [23].

### D. Browsers

Modern web browsers have modularized architectures that typically include rendering engines, JavaScript interpreters, and XML parsers [24]. By moving and changing these components, vulnerabilities in particular components can be mitigated. In this way, the browser itself and the underlying operating system can be protected. For example, the Cheetah browser [25] and the 360 browser [26] can change their rendering engines between WebKit and Trident.

Besides protecting browsers from exploits, changing browser configurations can also protect the privacy of web users. Every browser instance has its unique configurations, therefore web applications can uniquely fingerprint a browser in order to track users [27]. Diversifying a browser's font, plugins, and other configurations can prevent it from being fingerprinted, hence protecting the privacy of web users [28]–[30].

## IV. Source Code Language Diversification

To apply MTD ideas in the logic layer of the server-side, we propose to change the underlying language implementation of the web application; taking care to retain the main functionalities of the original application. In doing so, we prevent certain categories of vulnerabilities from being effectively exploited—remote code injection exploits would cease to work as code an attacker manages to insert will not match the web application's language. In addition, any unpatched or zero-day vulnerabilities present in the original language would not be available for exploit during the time frame of the randomization, as the language is completely different from what the attacker original perceives it to be.

In this section, we describe our implementation of a static MTD mechanism for the logic layer. As a first step, we choose to convert between PHP, a web development language used by approximately 82% of all web applications [31] and Python, which is used by popular companies such as Google, YouTube, Pinterest, and Bing [32], [33]. Our approach is to develop a translator to automatically convert a Python web application to PHP. We first translate the source code as-is, resulting in syntactically valid, but semantically invalid output in the target language.

We approached the problem of translating from Python to PHP by first exploring the available open source tools. However, no such tool exists, and we believe that this is due to the varying web application frameworks available for Python. While PHP is a programming language that was built for creating server-side web application logic, Python is a general-purpose language that can be used to write server-side web application logic. Therefore, there are many different frameworks for building server-side web applications in Python. For this reason, we focus on Python applications that utilize `cgi-lib`, however the our translation concept is general and can be implemented for other frameworks in the future.

Our translator is essentially a compiler, and to speed development we leverage existing functionality in Python to initiate the first step in translating to PHP—specifically, the use of Python's `ast` module to build an Abstract Syntax Tree (AST) of a Python program. Then, we use the Python

```
def _Print(self, t):
        self.fill("print ")
        do_comma = False
        if t.dest:
                self.write(">>")
                self.dispatch(t.dest)
                do_comma = True
        for e in t.values:
                if do_comma:self.write(", ")
                else:do_comma=True
                self.dispatch(e)
        if not t.nl:
                self.write(",")
```

Listing 1. Original _Print in unparse to generate Python code.

```
def _Print(self, t):
        self.fill("echo ")
        do_comma = False
        if t.dest:
                self.write(">>")
                self.dispatch(t.dest)
                do_comma = True
        for e in t.values:
                if do_comma:self.write(", ")
                else:do_comma=True
                self.dispatch(e)
        if not t.nl:
                self.write(",")
        self.write(";")
```

Listing 2. Modified _Print in unparse to generate PHP code.

`unparse` module, which is a module that converts an AST to Python code. We develop a new library based on `unparse` that generates PHP code instead of Python code.

Specifically, we modify the `unparse` module code by replacing the `print-to-output` function for a given Python statement with the corresponding PHP-specific statement. For instance, when translating a simple `print` statement from Python to the PHP equivalent of `echo`, we modify the `_Print` function in the `unparse` module as shown in Listing 1.

We replace the `print` Python keyword with the `echo` PHP keyword and ensure that the instruction is terminated with a semicolon as shown in Listing 2.

Once this is done, we have a program that is syntactically valid PHP, however it does not have the same semantics as the original Python program. Therefore, the next step is to make the translated program semantically equivalent to the original program. This step is necessary because there may not be a one-to-one translation for every feature in a language to another. For example, a Python instruction to terminate and exit the program is done using:

```
sys.exit(0)
```

After the translation is done and a PHP valid output is generated:

```
sys->exit(0)
```

However, PHP sees this as a new `sys` object with a call to a function `exit(0)`, which does not exist in PHP.

```
class sys
{
        <...constructor...>
        <...other functions...>

        public function exit($status)
        {
                exit($status)
        }
}
```

Listing 3. sys Object in PHP

The instruction does however have an equivalent function call—`exit(status)` in PHP. Therefore, we implement Python built-in functions as shims in order to match the new function calls. To this end, we create a PHP library that contains an object called `sys` with a function call to `exit(status)` as shown in Listing 3. This PHP library shim can be included in the translated application in order for the function call to remain semantically valid.

Using this approach, we recursively run the tool on the Python functions that the original program calls, and convert them as well. If, for instance, the original program is written in C, then we create a function shim for it.

However, in order to achieve the MTD goal, we must also decide on how frequently to move or randomize the component in order to be effective while considering the cost to legitimate users. Furthermore, there may be risk in the translated application missing critical function calls that we have not yet created shims. Finally, we anticipate this approach to be resource and time intensive as it is essentially creating two implementations of one web application.

## V. Database Dialect Diversification

To enable movement in the server-side storage layer, we implemented an approach to change the underlying database implementation, while preserving data and retaining functionality. In doing so, we again protect against certain categories of vulnerabilities and exploits—SQL injection exploits would be rendered ineffective due to syntactical differences between database implementations.

When performing the database translation, no alterations must be made to the data content—that is, once the translation is completed, the users must see the same information regardless of the underlying database implementation. Access to the database must be guaranteed and kept transparent to the user during and after the translation process. Database translations may be costly as well, especially regarding larger, more established databases—optimizations in the original implementation may become invalid once converted. Similar to our source to source approach, by continuously changing database implementations, we expect any database-specific exploits as well as unpatched or zero-day vulnerabilities will be ineffective.

As a first step, we choose to convert between MySQL and PostgreSQL, which are ranked 2nd and 5th in db-engines.com popularity ranking, respectively [34]. MySQL is used by well known companies, such as Facebook, Google, Amazon and Dropbox [35] and PostgreSQL is used by U.S. Dept of labor, U.S. State Department, and Sun Microsystems [36].

Some differences between the MySQL and PostgreSQL syntaxes include:

- The # or -- (A space after the -- is required) is used to begin a comment in MySQL, while PostgreSQL instead uses -- (the space is not required).
- Single (`''`) quotes or double (`""`) quotes are used in quoting values in MySQL, while PostgreSQL uses only single quotes for values, reserving double quotes in identifying field names, table names, etc.
- MySQL is case-insensitive when doing string comparision while PostgreSQL is case-sensitive, i.e. `john != JOHN != John`.

All of these differences affect the SQL injection exploits written to take advantage of an SQL injection vulnerability. If an attacker assumes that the web application is using a particular database backend, specifically if the attacker is searching the entire web for vulnerabilities, the exploit will fail.

Similar to our source-to-source approach, we developed a tool that can automate the conversion or migration between databases. In order to convert from PostgreSQL to MySQL, we modifed an existing open-source tool created by Lightbox that converts PostgreSQL to MySQL—although we can simply create a database dump from PostgreSQL, directly importing to MySQL will not work as there are differences between syntax and data types, which must be properly translated. In addition, certain flags need to be enabled when creating a database dump to allow for initial compatibility (PostgreSQL db dumps need to have `--inserts` enabled to properly include the data stored; MySQL needs to have `--compatible=postgresql` flag to properly include PostgreSQL keywords). To remedy this situation, our tool processes the original database dump by parsing the file and replacing any PostgreSQL keywords and data-types into corresponding MySQL keywords and data types. Some considerations have to made regarding conversions between data types, for instance PostgreSQL's `BYTEA` can be converted to any of the MySQL data types shown in Table I.

The data type chosen needs to be generic enough that it covers the possible data value that is in the original database, while attempting to be as performant as possible. When testing the original implementation of the converter, we observed that the output did not generate a database dump that is supported by the latest version of MySQL. In addition, it did not correctly convert the raw database dump from PostgreSQL, as the final output still contained PostgreSQL keywords and data-types. To handle conversion in the reverse direction, from MySQL to PostgreSQL, we re-purposed the code by reversing the process—that is, we parse through the dump file looking for MySQL keywords and data-types converting them to the corresponding PostgreSQL keywords and data-types.

| MySQL | PostgreSQL |
|---|---|
| BINARY(n) | BYTEA |
| VARBINARY(n) | BYTEA |
| TINYBLOB | BYTEA |
| BLOB | BYTEA |
| MEDIUMBLOB | BYTEA |
| LONGBLOB | BYTEA |

Table I
COMPARISON OF MYSQL AND POSTGRESQL DATA TYPES.

## VI. RELATED WORK

The idea and philosophy of moving target defense, which is to increase uncertainty and complexity for attackers, has been proposed and studied for decades [37]–[40]. Okhravi *et al.* surveyed techniques that applied the philosophy of moving target defense in different cyber research domains [41]. According to them, existing techniques can be categorized into five classes based on what-to-move: (1) changing runtime environment [12], [13], (2) changing application's code dynamically or diversifying software [14], [15], (3) changing data representations [38], [42], (4) changing platforms [16], [17], and (5) changing network configurations [43]–[45].

However, applying the moving target defense concept to web applications is still new. Huang *et al.* proposed to create and rotate among a set of virtual servers, each of which is configured with a unique software mix, to move attack surfaces for web surfaces [46]. Their work also explored the various opportunities of diversification in the web application software stack, providing a higher-level overview of the attack surface. Our work builds on this by further analyzing the components in each layer and defining what randomization in each layer entails, in addition to attempting an automated approach to diversification in the logic and storage layer. Boyd *et al.* proposed to create instances of unpredictable database query languages and to translate them to standard SQL using an intermediary proxy to prevent SQL injection attacks [47]. Although their approach also aims to prevent SQL injections, we chose a different approach in order to prevent a broader range of vulnerabilities—specifically unpatched vulnerabilities, zero day exploits, and mass-attacks targeting specific database implementations. Portner *et al.* proposed to defend cross-site scripting by mutating the symbols in JavaScript so that maliciously injected code can be identified [23]. Their work aims to prevent a different class of vulnerabilities, specifically located at the presentation layer on the client side. Our proposed approaches are aimed at applying MTD ideas on the server side of the web application architecture.

However, we envision techniques such as these, that are in each layer, to cooperate together to provide a defense-in-depth approach to defending web applications.

## VII. Future Work

As part of our future work, we plan to address the remaining semantic issues after translating from a Python application to a syntactically correct PHP application—one such issue is the translation of Python data structures to their equivalent in PHP; for instance, Python lists do not have a direct equivalent in PHP. Another component that we plan to explore is the possibility of automating the conversion to other web development languages. Similarly, we plan to explore the possibility of conversion to other database implementations. In addition, we will further investigate the other web application layers that can be moved.

At the heart of any MTD mechanism is the technique to decide when to move the chosen components. As such, we also plan to explore the various movement schemes and its effect on web applications that use our MTD approach. Finally, we will create a modular framework that can automatically apply our implemented MTD techniques in each layer of the web application in order to create a vast number of possible configurations. We will then evaluate this framework using real web applications deployed on a real-world network, in order to measure its effectiveness.

## VIII. Conclusion

In this paper we explored the feasibility of applying MTD concepts to web applications in order to create defensive layers. We analyzed the web application stack to understand where and how MTD can be applied, as well as current techniques implemented in each layer. In addition, we also discussed our implementation of two MTD approaches, which can mitigate several classes of web application vulnerabilities or exploits, wherein we change the language implementation of the web application and the database implementation while retaining functionality. We believe that MTD offers an exciting new research area in defending web applications, and we believe that the future of web application defense lies in reducing the information asymmetry inherent in the current web application security environment.

## Acknowledgment

This work was partially supported by the grant from National Science Foundation (NSF-SFS-1129561).

## References

[1] J. Silver-Greenberg, M. Goldstein, and N. Perlroth, "JPMorgan Chase Hacking Affects 76 Million Households," *The New York Times*, Oct. 2014.

[2] J. Robertson and M. Riley, "JPMorgan Hack Said to Span Months Via Multiple Flaws," Aug. 2014.

[3] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Saner: Composing static and dynamic analysis to validate sanitization in web applications," in *Security and Privacy, 2008. SP 2008. IEEE Symposium on*. IEEE, 2008, pp. 387–401.

[4] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna, "Toward automated detection of logic vulnerabilities in web applications," in *USENIX Security Symposium*, 2010, pp. 143–160.

[5] N. Jovanovic, C. Kruegel, and E. Kirda, "Static analysis for detecting taint-style vulnerabilities in web applications," *Journal of Computer Security*, vol. 18, no. 5, pp. 861–907, 2010.

[6] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna, "Enemy of the State: A State-Aware Black-Box Vulnerability Scanner," in *Proceedings of the USENIX Security Symposium (USENIX)*, Bellevue, WA, August 2012.

[7] A. Doupé, W. Cui, M. H. Jakubowski, M. Peinado, C. Kruegel, and G. Vigna, "deDacota: Toward Preventing Server-Side XSS via Automatic Code and Data Separation," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Berlin, Germany, November 2013.

[8] D. Wichers, "Owasp top-10 2013," *OWASP Foundation, February*, 2013.

[9] A. Cui and S. J. Stolfo, "Symbiotes and defensive mutualism: Moving target defense," in *Moving Target Defense*. Springer, 2011, pp. 99–108.

[10] R. Zhuang, S. A. DeLoach, and X. Ou, "Towards a theory of moving target defense," in *Proceedings of the First ACM Workshop on Moving Target Defense*, ser. MTD '14. New York, NY, USA: ACM, 2014, pp. 31–40. [Online]. Available: http://doi.acm.org/10.1145/2663474.2663479

[11] P. Ratanaworabhan, V. B. Livshits, and B. G. Zorn, "Nozzle: A defense against heap-spraying code injection attacks." in *USENIX Security Symposium*, 2009, pp. 169–186.

[12] P. Team, "Address space layout randomization," *Phrack*, 2003.

[13] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, and D. D. Zovi, "Randomized instruction set emulation to disrupt binary code injection attacks," in *Proceedings of the 10th ACM conference on Computer and communications security*. ACM, 2003, pp. 281–289.

[14] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "Sok: Automated software diversity," in *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 2014, pp. 276–291.

[15] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 157–168.

[16] D. Williams, W. Hu, J. W. Davidson, J. D. Hiser, J. C. Knight, and A. Nguyen-Tuong, "Security through diversity: Leveraging virtual machine technology," *Security & Privacy, IEEE*, vol. 7, no. 1, pp. 26–33, 2009.

[17] B. Salamat, T. Jackson, G. Wagner, C. Wimmer, and M. Franz, "Runtime defense against code injection attacks using replicated execution," *Dependable and Secure Computing, IEEE Transactions on*, vol. 8, no. 4, pp. 588–601, 2011.

[18] S. Vikram, C. Yang, and G. Gu, "Nomad: Towards non-intrusive moving-target defense against web bots," in *Communications and Network Security (CNS), 2013 IEEE Conference on*. IEEE, 2013, pp. 55–63.

[19] M. Dunlop, S. Groat, W. Urbanski, R. Marchany, and J. Tront, "Mt6d: A moving target ipv6 defense," in *MILITARY COMMUNICATIONS CONFERENCE, 2011 - MILCOM 2011*, Nov 2011, pp. 1321–1326.

[20] M. Carvalho and R. Ford, "Moving-target defenses for computer networks," *Security Privacy, IEEE*, vol. 12, no. 2, pp. 73–76, Mar 2014.

[21] Y. Li, R. Dai, and J. Zhang, "Morphing communications of cyber-physical systems towards moving-target defense," in *Communications (ICC), 2014 IEEE International Conference on*, June 2014, pp. 592–598.

[22] A. Doupé, B. Boe, C. Kruegel, and G. Vigna, "Fear the ear: discovering and mitigating execution after redirect vulnerabilities," in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 251–262.

[23] J. Portner, J. Kerr, and B. Chu, "Moving target defense against cross-site scripting attacks (position paper)," in *Foundations and Practice of Security*. Springer, 2014, pp. 85–91.

[24] C. Reis, A. Barth, and C. Pizano, "Browser security: lessons from google chrome," *Queue*, vol. 7, no. 5, p. 3, 2009.

[25] Liebao, "Cheetah browser. http://www.liebao.cn/index.html."

[26] Qihu, "360 browser. http://www.360safe.com/browser.html."

[27] P. Eckersley, "How unique is your web browser?" in *Privacy Enhancing Technologies*. Springer, 2010, pp. 1–18.

[28] P. Laperdrix, W. Rudametkin, and B. Baudry, "Mitigating browser fingerprint tracking: multi-level reconfiguration and diversification," in *Proceedings of the International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'15)*, 2015.

[29] N. Nikiforakis, W. Joosen, and B. Livshits, "PriVaricator: Deceiving fingerprinters with Little White Lies," in *Proceedings of the International World Wide Web Conference (WWW)*, 2015.

[30] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, "Cookieless Monster: Exploring the Ecosystem of Web-based Device Fingerprinting," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2013.

[31] Ide, "Php just grows & grows. http://news.netcraft.com/archives/2013/01/31/php-just-grows-grows.html."

[32] Sangster, "Organizations using python. https://wiki.python.org/moin/organizationsusingpython."

[33] Donohue, "Top 15 sites built with python. http://coderfactory.com/posts/top-sites-built-with-python."

[34] "Db-engines ranking. http://db-engines.com/en/ranking."

[35] "Mysql users. https://www.mysql.com/customers/."

[36] "Postgresql users. http://www.postgresql.org/about/users/."

[37] A. Avizienis and L. Chen, "On the implementation of n-version programming for software fault tolerance during execution," in *Proc. IEEE COMPSAC*, vol. 77, 1977, pp. 149–155.

[38] P. E. Ammann and J. C. Knight, "Data diversity: An approach to software fault tolerance," *Computers, IEEE Transactions on*, vol. 37, no. 4, pp. 418–425, 1988.

[39] K. Pettis and R. C. Hansen, "Profile guided code positioning," in *ACM SIGPLAN Notices*, vol. 25, no. 6. ACM, 1990, pp. 16–27.

[40] S. Forrest, A. Somayaji, and D. H. Ackley, "Building diverse computer systems," in *Operating Systems, 1997., The Sixth Workshop on Hot Topics in*. IEEE, 1997, pp. 67–72.

[41] H. Okhravi, M. Rabe, T. Mayberry, W. Leonard, T. Hobson, D. Bigelow, and W. Streilein, "Survey of cyber moving target techniques," DTIC Document, Tech. Rep., 2013.

[42] A. Nguyen-Tuong, D. Evans, J. C. Knight, B. Cox, and J. W. Davidson, "Security through redundant data diversity," in *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*. IEEE, 2008, pp. 187–196.

[43] R. Zhuang, S. Zhang, A. Bardas, S. DeLoach, X. Ou, and A. Singhal, "Investigating the application of moving target defenses to network security," in *Resilient Control Systems (ISRCS), 2013 6th International Symposium on*, Aug 2013, pp. 162–169.

[44] L. Ge, W. Yu, D. Shen, G. Chen, K. Pham, E. Blasch, and C. Lu, "Toward effectiveness and agility of network security situational awareness using moving target defense (mtd)," vol. 9085, 2014, pp. 90 850Q–90 850Q–9. [Online]. Available: http://dx.doi.org/10.1117/12.2050782

[45] J. H. Jafarian, E. Al-Shaer, and Q. Duan, "Openflow random host mutation: Transparent moving target defense using software defined networking," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, ser. HotSDN '12. New York, NY, USA: ACM, 2012, pp. 127–132. [Online]. Available: http://doi.acm.org/10.1145/2342441.2342467

[46] Y. Huang and A. K. Ghosh, "Introducing diversity and uncertainty to create moving attack surfaces for web services," in *Moving Target Defense*. Springer, 2011, pp. 131–151.

[47] S. W. Boyd and A. D. Keromytis, "Sqlrand: Preventing sql injection attacks," in *Applied Cryptography and Network Security*. Springer, 2004, pp. 292–302.