**Ubiquity Symposium**

# The Internet of Things

## Using Redundancy to Detect Security Anomalies: Towards IoT security attack detectors

***By Roopak Venkatakrishnan and Mladen A. Vouk***

### Editors' Introduction

*Cyber-attacks and breaches are often detected too late to avoid damage. While "classical" reactive cyber defenses usually work only if we have some prior knowledge about the attack methods and "allowable" patterns, properly constructed redundancy-based anomaly detectors can be more robust and often able to detect even zero day attacks. They are a step toward an oracle that uses knowable behavior of a healthy system to identify abnormalities. In the world of Internet of Things (IoT), security, and anomalous behavior of sensors and other IoT components, will be orders of magnitude more difficult unless we make those elements security aware from the start. In this article we examine the ability of redundancy-based anomaly detectors to recognize some high-risk and difficult to detect attacks on web servers—a likely management interface for many IoT stand-alone elements. In real life, it has taken long, a number of years in some cases, to identify some of the vulnerabilities and related attacks. We discuss practical relevance of the approach in the context of providing high-assurance Web-services that may belong to autonomous IoT applications and devices.*

 

# The Internet of Things

## Using Redundancy to Detect Security Anomalies: Towards IoT security attack detectors

***By Roopak Venkatakrishnan and Mladen A. Vouk***

Security attacks on Web-servers have increased by more than 30 percent from 2012 to 2013 [1]. Many of the attacks were successful despite extensive attempts to educate the software development community about the relevant vulnerabilities and exploits [2]. Well into 2015, the problem persists. In the world of "Internet of Things" (IoT), practically any device may have an IP address, an API, or a web interface and a certain level of "smartness." In the IoT—where devices are probably software-based, programmable, and autonomous—vulnerabilities and threats will continue to grow even further. In fact, the security problem will probably grow by orders of magnitude. Ideally, one would build security into these devices to make them inherently resilient. Unfortunately, in practice reliability and security of software-based system, sensors, and IoT elements can be improved only to a certain degree prior to deployment. Reasons are many, and may range from power consumption restrictions, to complexity, to poor designs or lack of planning.

Given the assumption that in normal operation such systems operate reliably, one of the questions is whether some "classical" reliability approach can help us develop after-market solutions that would make such systems more resilient to security attacks—or at least aware that they are under attack? Attacks usually change the behavior of a system. There are basically three ways of detecting execution-time anomalies or changes in behavior, and this includes security anomalies: acceptance testing, external consistency checking, and redundancy-based techniques.

Acceptance testing is a technique where the input to a system (or sensor) is validated against conditions specified by the developer or defender. This requires prior knowledge about the attack or malware patterns and signatures. On the other hand, external consistency checks work well at verification and validation time (e.g., fault-injection), but their applicability may be limited at run-time because the exact knowledge of the true conditions of operation may not be available to a system in the field. Finally, anomaly detection based on redundancy is flexible

 

and can be proactive but is often expensive (e.g., diversity based voters, error-coding). In one such approach, *N*-version programming, the same input is given to multiple functionally equivalent versions of the software. If all versions do not agree, there may be an issue. Anomaly detection techniques that work by comparing output vectors of diverse, but functionally equivalent, software have been in active use in the high-assurance software industry for a long time [3, 4]. Recently these techniques have been revisited in the context of cybersecurity for Web-based systems [5, 6].

There is a good chance IoT devices/systems will operate the same (perhaps newer and stripped down) types of Web servers as the ones used on servers today. A Netcraft analysis of Web servers in use today shows that more than 80 percent of websites use Apache, Nginx, or Microsoft (IIS) servers [7]. Therefore, it may be reasonable to assume, many of the attacks directed at Web servers will probably target one of the top three servers, or their derivatives.

In this article we examine some difficult-to-detect high-severity cyber attacks on Web services that have taken place in the last five years to see how a redundancy-based, possibly cloud-based, security anomaly identification mechanism might have made the detection of attacks (and of their effects) quicker and easier. The discussion that follows, however, applies to any IoT devices (sensors) so long as some diversity is available.

**Background: Anomaly Classification**

Attacks on Web servers can target a Web server directly or target applications it hosts, third party add-ons, and other components that are part of the Web service, such as database, scripting engine, etc. We limit our scope to attacks directed toward the Web server core. Of special interest is the ability of a comparison-based approach to detect zero-day attacks. In classifying anomalies and their detection we only consider those that manifest through the HTTP/HTTPS port of the Web server, and not its interactions with the host system or other applications.

Two categories of Web server exploits are of interest here (see Figure 1). Both could be perceived as zero-day exploits. One category targets inherent vulnerabilities that stem from issues in the Web server code, configuration, plug-ins and affiliated applications (for example PHP, CGI, or an SQL engine). Those are usually exploited by attacks through direct Web-interface channels (e.g., port 80). The other category are relatively “silent,” compromises that

 

may have happened indirectly (e.g., through an SSH attack). These we call lateral attacks and exploits.

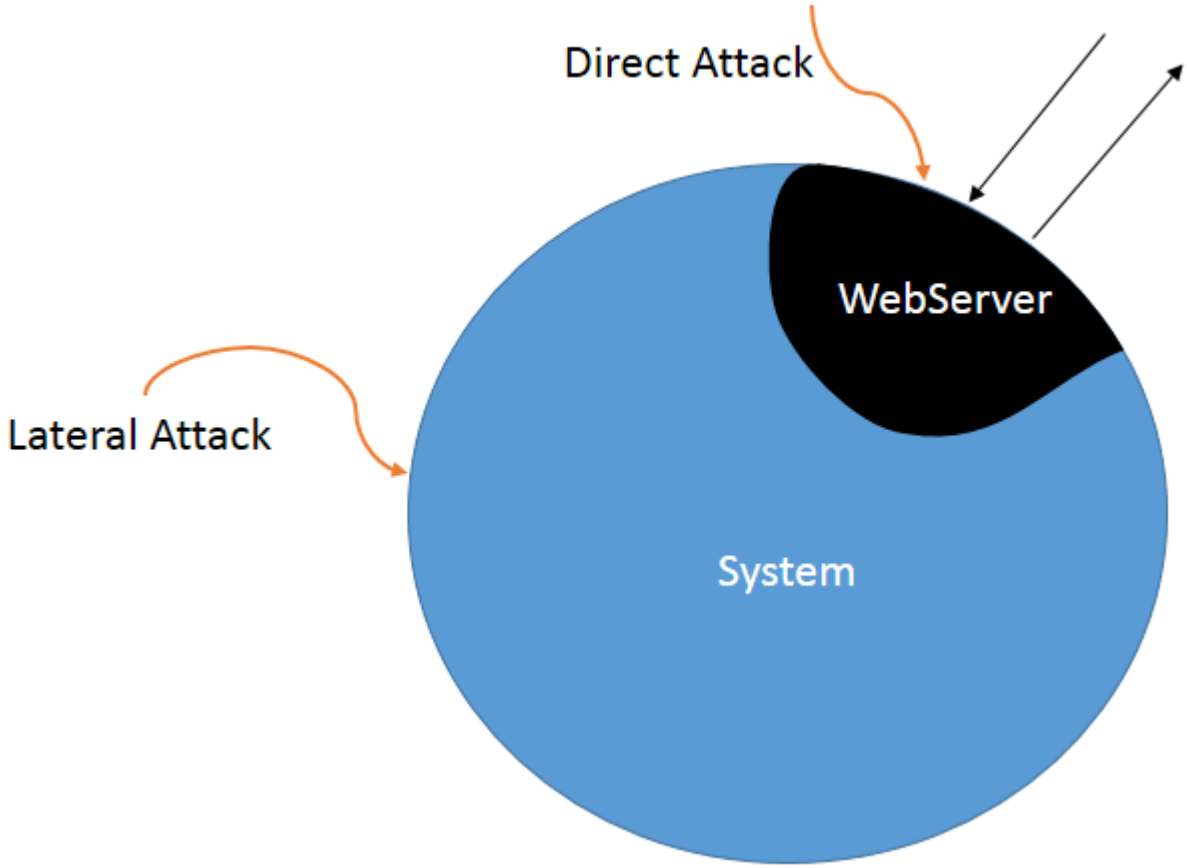


**Figure** 1. Types of attacks on Web servers.

Identifying oncoming attacks is often done quite successfully by acceptance tests either at the networking level, operating system level or application level (e.g., firewalls). However, acceptance tests do not work unless we have identified patterns or relationships that describe an observable characteristic of the attack, or its behavioral impacts. This usually means we need to experience an attack before we can defend against it.

An attack can result in:

- A modified server response but not a breach;

 

- An exploit, and as the result of that behavioral changes on its primary principal communication channel;
- Normal response despite being compromised.

**Failure Detection and Fault Tolerance**

In pro-active fault tolerance, failure is assumed and mitigation is always applied (e.g., error coding based forward recovery such as convolutional coding). In reactive fault tolerance, mitigation is applied only if an actual failure is suspected (e.g., memory checksum fails). In reactive fault-tolerance, the first step is to detect an anomaly.

Hardware can fail randomly as it gets older. In this context, using identical backup components (e.g., cold or hot standby) is a sensible approach. Several decades ago this idea was extended to the software space [8, 9, 10], but the problem is that software faults (and vulnerabilities) are usually due to design flaws or implementation errors not random wear, so replication may or may not guard against failures. For example, separate groups of developers could develop functionally equivalent components [10]. When different people are given identical specifications they may not use the exact same design and implementation details and algorithms. In fact, it is often assumed the faults left in such software components will not be the same, i.e., common-mode or correlated, meaning two functionally equivalent versions would not fail on the same input. A correlated failure is one where two or more components fail on the same input. If these failed components give the exact same response, the event is known as an identical-and-wrong response event [4]. Detection of such events is difficult using techniques that rely on the comparison of the outputs alone. A point to note is such a failure does not necessarily imply the versions failed due to a "common cause." Identical-and-wrong results could also be the result of "small output spaces" (e.g., system can only give results 0 and 1, the former being incorrect, then all incorrect results will be the same) [11]. A common cause fault is one where multiple versions fail because they may share a faulty piece of code or component or logic. The output of different versions in the case of a common cause failure could be similar but need not necessarily be identical.

This means everything from the language software is written in [12], to the operating system it runs on [13], to the run-time memory locations [14] can be used to make its run-time internals more diverse, and still keep its functional behavior the same. The idea is to make functionally equivalent software different in the right way, so the probability of correlated failures is as

 

close to zero as possible. Methods that manage random failures during software operation have been studied in great detail over the years [14, 15, 16, 17]. Diversity-based approaches work well as long as certain conditions (such as independence of failures among versions) are satisfied. In fact, redundant functionally equivalent software has been used in practice to, for example, fly aircraft [18] and guide trains [19].

In both acceptance-based algorithms and in comparison-based error-state detection algorithms one has to decide whether response(s) of a system is (are) acceptable. In the first case it is done against known patterns and behaviors, in the second case it is done against the behavior (or output) of the redundant versions. The algorithm that does the comparison and decides on this is known as an "adjudicator." There are many algorithms that are used for this purpose from simple acceptance tests (e.g., is the answer larger than X?), to majority voting, to median selection, to more complex comparison-based decision analysis. We distinguish between anomaly detection and actions (mitigation) that may follow adjudication.

**Effectiveness**

Redundancy-based anomaly detection is beneficial only if it is effective in detecting attacks, especially because of the increased cost of implementing it. Consider functionally equivalent but diverse versions $V_1$ , $V_2$ , . . . $V_i$ , . . . $V_N$ , $i$ = 1 . . . $N$. Let $T_j$ be the $j^{th}$ input vector (which may have a number of dimensions) that is received by all versions. Let there be $n$ input vectors (e.g., Web requests to Web server). Let the probability that version $V_i$ either fails (i.e., is compromised) or reacts to a security attack related probe in an anomalous fashion be $p_{Ai}$ ($T_j$ ). Let, for simplicity for all i and $i$ , $p_{Ai}$ ($T_j$ ) = $p$. If also we assume version failures or anomalous reactions are independent, i.e., $P$(version $i$ fails | version $k$ has failed) = $P$(version $k$ fails | version $i$ has failed) where $i \neq k$. Then it can be shown [4] that the probability that $N$ versions of the software detect an $n$-sequence attack stream is

$$P_D\,(n) = 1 - (1 - p)^{Nn} \qquad (1)$$

This scenario has very high probability of detecting almost any attempt to probe or intrude even for N = 3 or 2.

On the other end of the efficiency spectrum is scenario where all N versions have the same vulnerability. This basically means if one component fails, all may fail with an identical

 

response, and this exploit or probe cannot be detected. If the responses are not identical, there is a finite probability that the failures or anomalies stemming from this vulnerability or vulnerability probe will be detected. It can be shown [4] that in the first approximation, the probability that such a N-tuple detects such an attack is

$$P_D = 1 - [\gamma_N\, p + (1 - p)]^n \qquad (2)$$

where, $\gamma_N$ is the conditional probability that the *N* output vectors from the versions are accepted and are identical given that all the versions have failed.

A natural question in this context is "How often and how many Web servers (different versions or manufacturers) harbor an identical vulnerability"?

**Analyzing Vulnerabilities**

As part of this work we studied a number of major vulnerabilities or security issues found in the last five or so years in popular Web servers [6]. We investigated in-depth two most commonly used Web browsers [7]: Nginx and Apache. In the case of Apache, we investigated vulnerabilities in version 2.4. In some cases, the same issue was present in both server brands. The aim was to analyze comparisons between functionally equivalent but diverse servers as a potential vulnerability/attack detection mechanism. Table I summarizes the results.

**Table I.** Apache and Nginx vulnerabilities

| Web Server | Total Vulnerabilities | Major | Identifiable via comparison |
|---|---|---|---|
| **Nginx** | **17** | **9** | **4** |
| **Apache httpd** | **9** | **5** | **3** |

We see of the nine vulnerabilities studied in the case of Nginx four of them can be caught using a comparisons (or back-to-back testing [4]). Most of other vulnerabilities did not specifically show any change in output along the principal port of communication (Port 80 for HTTP and 443 for HTTPS). This means there is no difference that could be recognized through comparison

 

of principal port outputs from multiple versions. In the case of Apache, of the five vulnerabilities we studied, three were detectable using comparisons between server outputs. In other words, for the two Web server brands we studied, about half of the vulnerabilities could be detected using comparisons, while the other half remained undetected.

In addition to the vulnerabilities in the table, we also examined a few additional vulnerabilities that affected the Web servers, and caught the attention of the media. These additions were for most part lateral compromises that also included a few vulnerabilities in the Web server core. They were found by monitoring tech news sites like Hacker News [20] and blogs like Sucuri [21], and other outlets [22, 23, 24, 25, 26, 27].

Table II. Time taken to detect some recent web-server compromises.

| Attack | Type of Anomaly | Time Takento Detect | Estimated Insertion and Detection Dates | Impact (Number.of Servers) |
|---|---|---|---|---|
| Heartbleed Vulnerability (CVE-2014-0160) | Inherent | 2 years and 1 months | Mar 2012 / Apr 2014 | 600,000+ |
| Apache httpd (CVE-2014-0098), malformed cookie | Inherent | 8 years and 3 months | Dec 2005 / Mar 2014 | NA |
| Zencart Redirect – Feb 2014 | Inherent (Addon) | 1 year and 8 months | Jun 2012 / Feb 2014 | Unknown |
| Apache Tomcat DoS Attack (CVE-2014-0050) | Inherent | 3 years and 9 months | Jun 2010 / Feb 2014 | NA |
| Effusion - Modified Nginx Module 2013 | Lateral | Possibly 6+ years | Unknown/Dec 2013 | Unknown |
| Nginx Vulnerability (CVE-2013-4547) | Inherent | 3 years and 5 months | Jun 2010 / Nov 2013 | NA |
| Java.TomDep On Apache Tomcat 2013 | Lateral | Unknown | Unknown/Oct 2013 | 0-49[64] |
| Linux/Cdorked.A Attack 2013 | Lateral | 5 months | Dec 2012 / Apr 2013 | 20,000 websites |
| ColdFusion IIS Attack - (CVE-2013-0629) | Inherent | 3 years and 3 months | Oct 2009 / Jan 2013 | Unknown |
| Nginx Windows Restricted files (CVE-2011-4963) | Inherent | 3 years and 2 month | Apr 2009 / Jun 2012 | NA |
| Apache httpd (CVE-2011-3348)(malformed request) | Inherent | 2 years and 2 month | Jul 2009 / Sep 2011 | NA |
| Trojan.Apmod for Apache httpd 2011 | Lateral | Unknown | Unknown/Apr 2011 | 0-49[65] |
| IIS Bypass Authentication (CVE-2010-2731) | Inherent | 2 years and 5 months | Apr 2008 / Sep 2010 | NA |
| Apache httpd (CVE-2010-2068) (information disclosure) | Inherent | 2 years | Jun 2008 / Jun 2010 | NA |
| Nginx Windows Source Code (CVE-2010-2263) | Inherent | 1 year and 1 month | Apr 2009 / Jun 2010 | NA |
| Nginx Windows DoS via request (CVE-2010-2266) | Inherent | 1 year and 1 month | Apr 2009 / Jun 2010 | NA |
| IIS Access Restricted content WebDAV (CVE-2009-1535) | Inherent | 6 years and 1 month | Apr 2003 / May 2009 | NA |

Table II illustrates how long it has taken the community to detect or act on inherent vulnerabilities found in the Web server core and in add-ons, as well as some that identify a lateral exploit. In the case of inherent vulnerabilities the time taken is the time that an exploitable version has been out in the open without a patch available. All vulnerabilities listed

 

can be detected using back-to-back testing. On the average it would appear it has taken about three years or so to detect such vulnerabilities and exploits. By the time they are found considerable harm can be caused. Usually a difficult-to-find-vulnerability is detected only when it starts to affect a large number of public machines.

The number of inherent vulnerabilities appears to be much higher than that of lateral (or indirect) compromises. One reason is probably that lateral compromises may not use the main communication channel (e.g., port 80) for propagating the compromise. In that case, monitoring that channel may not reveal much. Unfortunately, laterally injected vulnerabilities appear to be more dangerous.

All vulnerabilities and attacks listed in Table II [6] show some change in outputs on the principal Web server port of communication. This is what makes them detectable using a diversity-based technique. Further, each vulnerability does not manifest in all Web servers simultaneously (e.g., only Nginx and Apache are affected, but MS IIS is not), or only in some particular environment (e.g., Windows only).

## Case Studies

The following cases illustrate how a triply redundant system could catch some of the (difficult) attacks. More details are available in [6].

### *Linux/Cdorked.A Attack 2013*

This is a situation where the Web server was compromised through a laterally injected exploit. This particular attack was, at the time, called the most sophisticated attack on Apache-like Web servers [28]. It is known to affect versions of Apache and Nginx. A recent Windingo report [29], indicates Cdorked was part of a larger operation, and a previous exploit on openSSH known as Ebury was used to deploy Cdorked. What is interesting is the malware was written specifically for each of Apache, Nginx, and Lighthttpd. Though much of the code was reused, the hooks for each version were different based on the targeted server.

*What Happened?*

Intrusion to inject Cdorked.A does not appear to leave any trace in the logs of the affected system, but Cdorked.A code is detectable if one knows where to look (or has a way of checking uncompromised installation logs). It always operates only in memory. Thus a simple restart

 

destroys all evidence of its operation. Initially the speculation was a flawed installation of add-ons from a compromised (fake) installer. The Windingo report [29] describes the method used by the attackers. A lateral compromise, which successfully broke into openSSH, gave the attackers access to deploy Cdorked. What made things worse is that the attack was designed in a way that made it very hard to detect by administrators of the machines. The infected Web servers had a plugin/module that sent the user a URL redirect based on conditions that included a random factor as well. Some of these conditions involved a check to see if the URL was from an admin-like page. If it was to not, the redirect happened. Since the malware was limited to in-memory operation, and Cdorked.A made sure that no information was stored in the logs, detection of Cdorked.A was a challenge.

*How can Redundancy Help?*

Our analysis, shows detection of this malware would have been much easier using back-to-back comparisons. Because it affected only Apache, and a small number of Nginx and Lighthttpd server versions, functionally equivalent server such as IIS remained unaffected.

Therefore, a triply redundant diverse system based on the Apache, Nginx, and IIS could have identified the compromise automatically by comparing results of the relatively simple combination of system inputs and outputs. The random factor built into the attack/compromise complicates the situation a bit. The same input query, when passed to multiple infected servers would only sometimes generate the malicious redirect. Therefore probability of redirect detection was lower, but this also meant even without the use of diverse (different) servers, redundancy would have been effective in detecting an unusual behavior. Use of different servers adds an additional advantage. Simple Web server status codes and response sizes could have been used to construct a back-to-back testing mechanism, and would have raised an alert provided a sufficient number of requests were given to the system.

Obviously it is possible to experience false alarms. However, multiple such alerts should indicate that something is wrong. This is better than what was actually experienced with this malware. Web server administrators did not know their servers had been compromised in some cases for over six months [26]. Cdorked.A had been initially noticed in August 2012, but without much idea of what exactly it was. ESET and Sucuri analyzed and wrote a script to detect this rogue module in early February 2013 when the attack gained momentum [29]. Several thousand servers were infected even after this, as it was only on March 20th and later that anti-virus scanners added this to their databases.

 

This particular attack, affected both Apache and Nginx. It is possible that an attack could affect all types of servers a diversity-based approach uses. But, even in such a case use of three compromised Apache and Nginx servers may have potentially detected the attack. The trigger for redirection included a random factor, which makes the probability of identical-and-wrong outputs less than one and gives a chance of detecting this particular attack even with compromised servers so long as random events to not coincide.

It is also important to note in all situations discussed herein, it is assumed that the output comparator is not compromised (i.e., that it does not "lie"). That can be achieved in a number of ways. For example, protected (read-only) code, independent (remote, perhaps cloud) processing, and similar.

***An Nginx Vulnerability (CVE-2013-4547)***

In November 2013, an engineer from Google discovered a vulnerability in Nginx [30]. This vulnerability could have allowed an attacker to bypass security restrictions. This is an inherent fault in the software, and it is exploited through specially crafted server communication port requests. Again a comparison based system could help detect this attack.

*What Happened?*

When a request is passed to a vulnerable Nginx (versions 0.8.41 through 1.4.3 and 1.5.x before 1.5.7) checks were not performed on unescaped space characters. The space character is an invalid character according to the HTTP protocol [31, 32, 33]. This was permitted for compatibility reasons. This allowed for certain security restrictions to be bypassed in some cases. For example, his made it possible to access a directory with permissions set to "denyall." [6] This exploit needed a tool like Telnet, which sends the HTTP request as is without modifying or escaping characters. According to the error report, this flaw was released sometime in 2010. Three years to discovery/patch is quite a while, considering millions of Web servers through the world use Nginx. Fortunately, this vulnerability may not have been exploited too frequently because of its pre-requisite requirements. Furthermore,

*How can Redundancy Help?*

Having a back-to-back comparison option in place could have detected this potential zero-day attack. Consider a very simple system. Let us say the system has three Web servers sensors and an independent and protected comparator. Let one of these be Nginx based, and that the other two be Apache and IIS based.

 

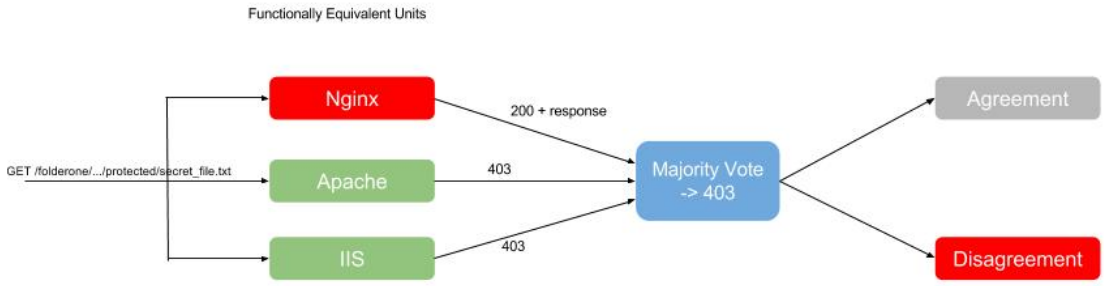


**Figure 2.** Comparison could be used to detect the Nginx vulnerability and decide on the correct response.

Figure 2 illustrates the approach. The input is the attack Web request. All Web servers are passed the same request. Let's assume the comparator looks only at the status codes. In this example, only Nginx is vulnerable and it returns the 200 status code (OK) and the response itself. However, the other two servers return a 403 (Forbidden), as they should. In this case Nginx automatically causes a disagreement, and hence an alert. In this example, two version redundancy would have detected the issue. Alert mitigation would have to be considered separately.

***The Heartbleed Vulnerability - CVE-2014-0160***

A vulnerability in OpenSSL was reported on April 7, 2014 by Neel Mehta at Google. This particular vulnerability had large potential impact. It is thought to have left two-thirds of the servers on the Internet vulnerable [34]. At least half a million servers were still exposed on April 10$^{th}$ (three days after the exploit was disclosed). The attack does not leave an obvious trace, and can provide an attacker with information such as private keys, or even user account passwords [22, 35]. It was caused by a simple implementation error.

*What Happened?*

In SSL, an extension called heartbeat keeps a session alive provided both parties want to keep the session alive even though they currently have no data to exchange. It consists of a payload and a matching response to make sure that the connection is functional. The payload is supposed to be limited to 16KB according to RFC 6520 [36]. However, the vulnerable implementations had a missing bound check on the size of the payload. Since the response is

 

supposed to contain the original payload sent to the server, the server copies this back for the response. Let a small payload, say one byte, be sent, but let the length of the payload be set to 65,535 bytes. With the bug present, the server does not check the size, and copies back into response to client 65,535 bytes. This means the library copies from the memory segments it was not meant to read. This allowed attackers to get passwords and private key information [35].

*How can Redundancy Help?*

This is a case of an inherent attack, but the problem is with a third party add-on, and not directly with the Web server. Diversity could have been used to detect the first attempt at exploit. All the information exchange takes place over port 443 (or the primary port for communication of a web- server when using HTTPS). This being the case, if a diversity-based setup were to inspect all traffic on this port, and not just basic requests and responses (not just the GET and POST), this attack would be very simple to catch. An improper implementation of SSL would return more than 16KB of data whereas any other implementation would fail without a response. In this case diversity would have to involve servers not using OpenSSL. For example, Microsoft services utilize SChannel as opposed to openSSL and thus anything using a Microsoft based stack remains unaffected [37].

**Discussion**

Some of the vulnerabilities in Table II could be detected just by looking at the size of the responses from the Web server. Some would require analyzing the output from each of the Web servers while some would be caught if the logs for each request were analyzed. The heartbleed vulnerability showed not only every request to the server should be analyzed to some extent, but every input over the principal port of communication (443 in the case of HTTPS) should be compared.

Figuring out how many parameters from the various output vectors need to be compared to detect most of the attacks is a question that needs to be answered as part of the comparator design. Our work [6] indicates that an output vector that has at least five different parameters may be sufficient to catch practically all attacks we have examined. However it should be noted comparing longer output vectors carries a cost and complexity penalty.

 

For redundancy to make economic sense, the increase in the cost needs to be justified by the need for an increased ability to detect attacks. In the case of an IoT-based system that may be operating for months and years without any direct administrative intervention or oversight, this investment may be worthwhile.

The voting based algorithms and solution proposed by Totel et al. [5] is a redundancy and diversity based solution intended to catch security vulnerabilities in static Web server content situations. The authors observe that for their architecture to work with dynamic content, such as scripts and databases, replication would need to be at many levels and each component would be dealt with individually. The following have to be considered when trying to implement a redundancy based system.

**Masking design differences**. When multiple versions of software are designed to do the same thing, they are all designed according to a specification. There will always be some parts where the specifications are not exact. At this point it is usually up the developers to decide how to do it. In the case of Web servers, one would probably have multiple functionally equivalent COTS (commercial of the shelf) components developed by different manufacturers. This means each group of developers may have implemented the less fully specified portion of the software in a different manner. To compare the outputs from such software, we may need to mask the design differences.

A simple example would be URL terminating with a "forward slash," It is treated differently on different Web servers. If a directory is requested, a terminating forward slash does not make a difference in case of the request. However, servers may handle a request with a slash slightly differently than the one without it.

So before the request is passed on to each Web server, steps need to be taken to mask design differences. This might have to be done by modifying the request before the request is passed to the Web server, or by normalizing output after the response is received from the Web server [5, 11].

**Costs.** Even if a system can detect and prevent 100 percent of the attacks that comes its way, it is not a viable solution unless it is affordable. Redundancy requires the use of extra resources. In IoT systems, particularly large autonomous ones—such as cars of the future, autonomous farm equipment, or house alarm systems, etc.—this may be a justifiable and in fact required expense. Furthermore, cloud computing may help a lot in this context. In the cloud it is very easy to spin up an instance and bring down an instance as required. Hence in the case we

 

detect an attack, and we know which server or sensor may have been compromised, the system can be brought back to a safe state and the detection process restarted. This would typically involve bringing up a new virtual machine, rolling forward so that all the Web servers are in sync and in the same state. This process is achieved very easily in the cloud. In a field deployed IoT sensor situation the cost of operating redundant sensors may not be high, but recovery from an anomaly may be very situations specific.

## Conclusions And Future Work

We have shown diversity-based detectors may help identify attacks that otherwise cannot be detected easily, including quite a few zero-day attacks. In this day and age of cloud computing, and thus perhaps readily available functionally equivalent services, such an approach may make economic sense [9]. In the IoT world, each component has an IP and an API. IoT units may be in the field for months and years without much administrative oversight. In fact, given the diverse nature of IoT elements and their scale, scope, and speed the securing them is orders of magnitude more complex than security more monolithic entities of today. Having redundancy, just to increase reliability and preserve functionality of IoT elements makes sense. Leveraging diverse redundancy to provide security awareness seems like a natural extension.

Diversity based attack detectors need to be secure and need to have low false positive rate to be practical. For example, an anomaly detection system that identifies 30 percent of the inputs as an attack when the actual number of attacks is only about two or three percent may not be very useful. Mitigation of detected attacks is a separate question.

## Acknowledgments

This work is supported in part through NSF grants 0910767, 1330553, and 1318564, the U.S. Army Research Office (ARO) grant W911NF-08-1-0105, the NSA NCSU Science of Security Lablet, and by the IBM Shared University Research and Fellowships program funding.

 

**References**

[1] Symantec. 2013 Internet Security Threat Report, volume 18. 20013.

[2] The Mitre Corp. CWE/SANS Top 25 Most Dangerous Software Errors. Sept. 13, 2011.

[3] D. McAllister and M. Vouk. Fault-tolerant software reliability engineering. In *Handbook of Software Reliability Engineering*. McGraw Hill, Hightstown, NJ, 1996, 567–614.

[4] M. A. Vouk. Back-to-back testing. *Inf. Softw. Technol.* 32, 1 (Jan. 1990), 34–45.

[5] E. Totel, F. Majorczyk, and L. Me. COTS diversity based intrusion detection and application to web servers. In the *Proceedings of the 8th International Conference on Recent Advances in Intrusion Detection* (RAID'05). Springer-Verlag, Berlin, Heidelberg, 2006, 43–62.

[6] R. Venkatakrishnan. Redundancy-Based Detection of Security Anomalies in Web-Server Environments. North Carolina State University. M.S. thesis, 2014.

[7] Netcraft. October 2015 web server survey. Oct. 16, 2015.

[8] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering* 1, 2 (June 1975), 220–232.

[9] A. Avizienis and J. P. J. Kelly. Fault tolerance by design diversity: Concepts and experiments. *Computer* 17, 8 (Aug. 1984), 67–80.

[10] D. Eckhardt, A. K. Caglayan, J. Knight, L. D. Lee, D. McAllister, M. Vouk, and J. Kelly. An experimental evaluation of software redundancy as a strategy for improving reliability. *IEEE Transactions on Software Engineering* 17, 7 (July 1991), 692–702.

[11] D.F. McAllister, C.E. Sun, and M.A. Vouk. Reliability of voting in fault-tolerant software systems for small output spaces. *IEEE Trans. Rel*. 39, 5 (1990), 524-534.

[12] A. Avizienis, M. R. Lyu, and W. Schultz. In search of effective diversity: a six-language study of fault-tolerant flight control software. In the *Eighteenth International Symposium on Fault Tolerant Computing* (FTCS-18). IEEE, Washington D.C., 1988, 15–22.

 

[13] M. Garcia, A. Bessani, I. Gashi, N. Neves, and R. Obelheiro. Analysis of operating system diversity for intrusion tolerance. *Software: Practice and Experience* 44, 6 (2014), 735–770. DOI: 10.1002/spe.2180.

[14] H. Shacham, M. Page, B. Pfaff, Eu-Jin Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security* (CCS '04). ACM, New York, 2004, 298-307.

[15] M. R. Lyu. *Software Fault Tolerance*. John Wiley & Sons, New York,1995.

[16] M. R. Lyu et al. *Handbook of Software Reliability Engineering*. McGraw Hill, 1996.

[17] J.-C. Laprie,. *Dependability: Basic concepts and terminology*. Springer, 1992.

[18] Y. Yeh. Design considerations in Boeing 777 fly-by-wire computers. In the *Third IEEE International Symposium on High-Assurance Systems Engineering*. IEEE, Washington D.C., 1998, 64–72.

[19] H. Kantz and C. Koza. The ELEKTRA railway signaling system: Field experience with an actively replicated system with diversity. In the *Twenty-Fifth International Symposium on Fault-Tolerant Computing*. IEEE, Washington D.C., 1995, 453–458.

[20] Ycombinator. Hacker News.

[21] Sucuri Inc. Sucuri Blog.

[22] B. Schneier. Heartbleed. Schneier On Security. Blog. April 9. 2014.

[23] D. Sinegubko. Mysterious zencart redirect leverages HTTP headers. Sucuri Blog. Feb. 16, 2014.

[24] L. Constantin. Cyber criminals offer malware for Nginx, Apache Web servers. ComputerWorld. Dec. 24, 2013.

[25] Symantec. Java.Tomdep. Security response. 2013.

[26] D. Goodin. Ongoing malware attack targeting Apache hijacks 20,000 sites. ArsTechnica. April 2, 2013.

[27] Symantec. Trojan.Apmod. Security response. 2011.

 

[28] ESET. ESET and Sucuri uncover Linux/Cdorked.A: The most sophisticated Apache backdoor. Press release. April 29. 2013.

[29] O. Bilodeau, P.-M. Bureau, J. Calvet, A. Dorais-Joncas, M.-E. M. Léveillé, and B. Vanheuverzwijn. Operation Windingo – The vivisection of a large Linux server-side credential stealing malware campaign. White paper. ESET. March 2014.

[30] Common Vulnerabilities and Exposures (CVE). Vulnerability in NGINX, CVE-2013-4547. 2013.

[31] J. Starr. Stop using unsafe characters in URLs. Perishable Press. Dec. 31, 2012. Updated Nov. 3, 2015.

[32] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifier (URI): Generic syntax. RFC 3986. IETF. Network Working Group. Jan. 2005. © The Internet Society.

[33] T. Berners-Lee, L. Masinter, and M. McCahill. Uniform Resource Locators (URL). RFC 1738. IETF. Network Working Group. Dec. 1994.

[34] D. Goodin. Critical crypto bug in OpenSSL opens two-thirds of the Web to eavesdropping. ArsTechnica. April 7, 2014.

[35] D. Goodin. Critical crypto bug exposes Yahoo Mail, other passwords Russian roulette-style. ArsTechnica. April 8, 2014.

[36] P. Ducklin. Anatomy of a data leakage bug - the OpenSSL “heartbleed” buffer overflow. Naked Security. April 8, 2014.

[37] L. Tung. Google, AWS, Rackspace affected by Heartbleed OpenSSL flaw - but Azure escapes. ZDNet. April 10, 2014.

**About the Authors**

Roopak Venkatakrishnan received his bachelor’s degree in computer science from Anna University, Chennai in 2012 and a master’s in computer science from North Carolina State University in 2014. He is currently a software engineer at Twitter.

 

Dr. Mladen Alan Vouk is a distinguished professor and department head of computer science. He is also director of the North Carolina State Data Science Initiative. Dr. Vouk has extensive experience in both commercial software production and academic computing. He is the author/co-author of more than 300 publications. His interests include software and security engineering, bioinformatics, scientific computing and analytics, information technology assisted education, and high-performance computing and clouds. Dr. Vouk is a member of the IFIP Working Group 2.5 on Numerical Software, and a recipient of the IFIP Silver Core award. He is an IEEE Fellow, and a recipient of the IEEE Distinguished Service and Gold Core Awards. He is a member of several IEEE societies, and of ASEE, ASQ (Senior Member), ACM, and Sigma Xi.

**DOI:** 10.1145/2822881

 