Security Risks in Asynchronous Web Servers: When Performance Optimizations Amplify the Impact of Data-Oriented Attacks

Micah Morton,* Jan Werner,* Panagiotis Kintis,† Kevin Snow,‡
Manos Antonakakis,† Michalis Polychronakis,§ Fabian Monrose*

*University of North Carolina at Chapel Hill; email: {micah,jjwerner,fabian}@cs.unc.edu,
†Georgia Institute of Technology; email: {kintis,manos}@gatech.edu,
‡Zeropoint Dynamics; email: kevin@zeropointdynamics.com,
§Stony Brook University; email: mikepo@cs.stonybrook.edu

***Abstract*—Over the past decade, many innovations have been achieved with respect to improving the responsiveness of highly-trafficked servers. These innovations are fueled by a desire to support complex and data-rich web applications while consuming minimal resources. One of the chief advancements has been the emergence of the *asynchronous* web server architecture, which is built from the ground up for scalability. While this architecture can offer a significant boost in performance over classic forking servers, it does so at the cost of abandoning memory space isolation between client interactions. This shift in design, that delegates the handling of many unrelated requests within the same process, enables powerful and covert data-oriented attacks that rival complete web server takeover — without ever hijacking the control flow of the server application.**

To demonstrate the severity of this threat, we present a technique for identifying security-critical web server data by tracing memory accesses committed by the program in generating responses to client requests. We further develop a framework for performing live memory analysis of a running server in order to understand how low-level memory structures can be corrupted for malicious intent. A fundamental goal of our work is to assess the realism of such data-oriented attacks in terms of the types of memory errors that can be leveraged to perform them, and to understand the prominence of these errors in real-world web servers. Our case study on a leading asynchronous architecture, namely Nginx, shows how data-oriented attacks allow an adversary to *re-configure* an Nginx instance *on the fly* in order to degrade or disable services (*e.g.,* error reporting, security headers like HSTS, access control), steal sensitive information, as well as distribute arbitrary web content to unsuspecting clients — all by manipulating only a few bytes in memory. Our empirical findings on the susceptibility of modern asynchronous web servers to two well-known CVEs show that the damage could be severe. To address this threat, we also discuss several potential mitigations. Taken as a whole, our work tells a cautionary tale regarding the risks of blindly pushing forward with performance optimizations.

1. Introduction

Since the earliest memory corruption attacks emerged as serious threats to the security of computer systems, security professionals have been tirelessly trying to stay ahead of exploitation tactics. Much of this defensive effort has focused on thwarting attacks that corrupt application control structures in order to hijack the execution of running software. Data Execution Prevention (DEP), which enforces that writeable data sections of a program (*e.g.,* the stack) are not also executable, and Address Space Layout Randomization (ASLR) are two prominent examples of widespread defenses that have been incorporated into mainstream systems. However, these defenses were later shown to be less effective than first thought given a single memory disclosure [33].

Accepting the fact that there will be exploitable bugs in complex programs, the designers of modern browsers have chosen to limit exploitation by delegating buggy rendering code to unprivileged *sandbox* processes. Similarly, contemporary web servers are built in a way that delegates connection parsing and processing to lower-privilege *worker* processes. In both cases, these design decisions force adversaries to further employ privilege escalation attacks to gain system-level access, which in turn adds an extra layer of sophistication in order to successfully exploit an application. While not perfect, these mitigations significantly raise the bar for control-hijacking attacks.

That being said, as system compromise through control flow hijacking becomes more difficult due to the myriad of defenses that have been deployed in this space, adversaries will undoubtedly explore new paths of least resistance. One such path is via the so-called *data-oriented attacks* that leverage the power of memory corruption to target non-control data for the purpose of exploiting applications without ever corrupting control flow [11, 21, 22, 23, 30].

We take a multi-step approach in demonstrating the feasibility of data-oriented attacks against modern web servers. We show that these attacks are made easy because of performance versus security tradeoffs that have been made by web server architectures. To elucidate these issues, we first describe a method for locating security-critical configuration

DOI 10.1109/EuroSP.2018.00020

data structures by tracing server execution during request processing. We then propose an automated framework for live memory analysis which can be used to expose the low-level state of critical data structures at runtime, matching different live memory states with different configuration file parameters on disk. Next, we show how our automated framework can be used to produce *faux* copies of key server data structures without any need for manual source code analysis or reverse engineering. Using this framework, we demonstrate how an adversary can leverage real-world memory disclosure and corruption vulnerabilities to *re-configure* a running web server *on the fly*, by redirecting data pointers to *faux* structures, instead of redirecting code pointers to malicious code. We present a complete case study of such data-oriented attacks against the contemporary Nginx web server, and evaluate the covertness of our demonstrated attack in the face of common real-world security-hardened deployment scenarios. Our specific innovations include:

- A flexible and robust instrumentation technique for identifying security-critical data in web server memory.
- An approach for bypassing ASLR using only a linear heap memory disclosure vulnerability.
- Highlighting how an adversary can significantly reduce the work factor involved in server takeover (compared to what is typically considered necessary using contemporary approaches).
- Evaluating the feasibility of such attacks by studying the widespread susceptibility of deployed web servers to vulnerabilities that enable such attacks.

2. Background

Although modern web servers generally carry out a set of straightforward tasks when handling incoming requests (*e.g.,* accepting network connections, parsing client requests, fetching content from a datastore, and generating responses), there have been a number of proposed approaches to implementing this workflow. The differences can be attributed to varying standards for scalability, performance, robustness, and simplicity in design. Designing a web server architecture that is optimized for any of these high-level attributes involves awareness of how to leverage lower-level operating system features (*e.g.,* processes, threads, asynchronous I/O).

One approach relies on using a different process or thread for each connection being serviced. This greatly improves the scalability of servicing requests through synchronous I/O, since the process or thread associated with a given request can be suspended while waiting for an I/O operation to complete — freeing resources which can be dedicated to processing additional requests. In recent years, this model has been popularized by the Apache web server, which forks a separate process to handle each incoming connection, terminating it upon connection closure. One notable optimization of Apache's process-per-request architecture involves *preforking* a pool of processes on startup to avoid the overhead of forking upon each incoming connection. While using multiple processes for handling concurrent requests indeed benefits scalability, the heavyweight nature

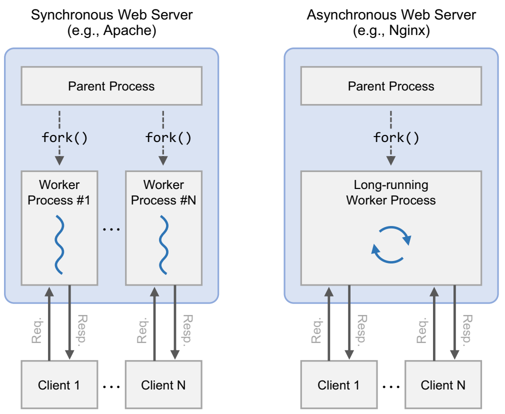


Figure 1: Synchronous vs. asynchronous web servers.

of a process object, as well as the overhead of context switching between processes, means that this model is not satisfactory for web servers that must handle hundreds or thousands of incoming connections concurrently.

In response to demands for highly concurrent web servers, traditional process-based architectures such as Apache have begun to offer thread-based concurrency that allows a single process to service multiple concurrent connections by dedicating a unique thread to each connection. In this way, one thread in a process can block while waiting for an I/O operation to complete at the same time that other threads continue to service other requests. This approach, called *worker* mode by Apache [4], is a popular alternative to process preforking when scalability to many connections is important, but allocating a thread for each connection is still considered inefficient for many real-world servers [24].

As the demand for web server concurrency has increased, a new architecture emerged: the asynchronous (event-driven) web server. Under this model, requests are serviced asynchronously by a single (single-threaded) worker process, which uses event-based callback functions to carry out server functionality when needed (*e.g.,* parse request headers, construct response headers). Since blocking on synchronous I/O is not necessary, connections do not need to be associated with a scheduling unit that can be suspended, providing greater scalability. Note that the functionality that enables asynchronous request processing (*e.g.,* chaining processing modules together via callback functions) must be at the core of the overall server architecture and must be incorporated into many design aspects.

Despite the challenges of refactoring its core synchronous processing implementation, Apache recently offered a processing mode known as *event* [4], which makes further strides to optimize the number of clients that can be handled simultaneously by a single worker process. As we show later, the risks of abandoning web server memory space isolation between client requests, will only become more relevant as Apache continues to refactor its server design to match the impressive scalability performance offered by asynchronous architectures.

Nginx (pronounced *engine-x*), the market's most popular asynchronous web server, has garnered widespread adoption as a result of its ground-up design for asynchronous scalability [39]. In fact, although Apache still holds the largest market share, many sites have switched to Nginx in recent years (potentially also incorporating other back-end processing solutions). At the start of the decade in 2010, Apache claimed 71.5% of the web server market, while Nginx was only used by 3.9% of sites. However, as of January 2017, only 50.9% of sites still use Apache, while 32.1% use Nginx. The popularity of Nginx is especially apparent for the busiest websites, as the majority of the busiest 100,000 sites use Nginx over Apache [41].

Figure 1 shows the high-level architectural differences between the industry's two most popular web servers. Critically, the figure shows the difference in how Apache uses process-based isolation to logically separate request processing, while Nginx handles all requests in a single process. This key difference in architectural models has major implications in terms of the susceptibility of these web servers to non-control-data oriented attacks.

2.1. Exploiting Web Servers

Exploiting a web server can be a desirable feat for mounting widespread attacks against unsuspecting clients. Web server exploitation is often the first step in a drive-by download campaign, where the ultimate goal is to use the popularity of a legitimate website to distribute malware once the web server has been compromised. To put the findings of this work in perspective, it is important to understand the requirements for a modern-day *exploit chain* that seeks to gain system level control of a victim machine. Due to ubiquitously deployed mitigations such as DEP and ASLR, full system exploitation generally requires an adversary to:

1) Exploit a memory corruption vulnerability to modify the contents of an application's memory.
2) Leverage a memory disclosure bug to circumvent address space randomization.
3) Prepare a code re-use payload in memory and pivot the stack pointer to the start of this chain.
4) Use the ROP chain to map the location of injected shellcode as executable.
5) Launch a privilege escalation attack against higher-privilege components.

Each of these steps in the exploit chain provide unique challenges to an adversary. In particular, accepting the fact that memory errors will inevitably occur in complex applications written in type-unsafe C/C++ code, the research community has focused heavily on raising the bar for steps 3–5 through DEP and code reuse defenses, sandbox development, kernel hardening and many others.

Interestingly, while the absence of untrusted script execution protects web servers from many associated vulnerabilities, the non-trivial logic implementing complex request processing and dynamic content generation exposes a considerable attack surface to adversaries. Indeed, Hu et al. [22] recently showed the feasibility of achieving *arbitrary write* capabilities against popular server programs, thereby confirming the generally accepted notion that motivated adversaries will find ways to leverage memory corruption exploits (*e.g.,* buffer overflow, use-after-free, double free) in order to achieve the so-called *write-what-where* capabilities [26]. This scenario — which affords the ability to write an arbitrary value at an arbitrary location in process memory — can be enacted in a variety of ways, such as corrupting stack or heap objects that will be written to in the future. Like Hu et al. [22], we assume the existence of an arbitrary write vulnerability in Nginx for the proof of concept exploits presented in Section 6.

Although we assume such *arbitrary write* capabilities, we do *not* assume the ability to use memory corruption to gain *arbitrary read* capabilities. In particular, after extensive research, we found no practical exploits or exploit methodologies that can be leveraged to disclose server memory at an arbitrary address. Although such exploits may exist, we restrict ourselves from asserting the theoretically powerful assumption of arbitrary read capabilities due to their rarity and to keep with our goal (§4) of presenting attacks that are feasible in the real world.

On the other hand, there have been instances of server vulnerabilities that disclose a *linear* swath of heap memory (*e.g.,* Heartbleed (CVE-2014-0160), Cloudbleed [18], Yahoobleed (CVE-2017-9098), CVE-2014-0226, CVE-2012-1180) at an unspecified address. The Heartbleed vulnerability, for example, was one of the most impactful security issues in the last decade, with 24–55% of HTTPS servers in the Alexa Top 1 million sites being initially vulnerable [14]. In early 2017, researchers uncovered the Cloudbleed vulnerability in Cloudflare's CDN service, due to a memory error in an Nginx module used for parsing and modifying HTML pages on-the-fly [18]. This vulnerability serves as a reminder that complex and memory-error-prone processing is employed by cloud-based services within the confines of Nginx's asynchronous architecture. While Heartbleed, Cloudbleed, and similar vulnerabilities do not give the adversary as powerful of a primitive as *arbitrary read*, we show that even a partial *linear read* of heap memory (whose location is not controlled by the adversary) can be leveraged to undermine ASLR and locate key application structures as a first step in performing powerful data-oriented attacks.

3. Other Related Work

Over a decade ago, Chen et al. [11] highlighted the power of leveraging memory corruption exploits to subvert systems through the manipulation of security-critical non-control-data — all without ever corrupting the control flow structures of an application. They demonstrated data-oriented attacks against an assortment of widely-used server-side applications, but their approach required manual source code analysis to obtain in-depth semantic knowledge regarding the layout of security-critical data and how its corruption could be leveraged in each application. More recently, Hu et al. [21] showed how to lessen the amount of a-priori knowledge needed for pulling off the same attacks presented by Chen et al. [11]. Their approach, termed *data-flow stitching*, utilizes taint tracking to compute data flows that occur during application runtime. This approach treats file inputs to the application as data sources and file outputs as data sinks, tracing how critical data is imported to an application from the file system as well as how information generated by the program flows out to the filesystem. Shortly thereafter, Hu et al. [22] highlighted the feasibility of using commonly occurring memory corruption vulnerabilities to gain arbitrary write capabilities in server programs. That work shows how memory errors can be leveraged to achieve *write-what-where* [26] capabilities in process memory.

None of these works provide a general technique for overcoming ASLR, but rather require that a pointer to security-critical data is somehow leaked to the adversary by the same memory error that allows for the arbitrary write. Thus, it is unclear how an adversary would adapt the opaque payloads generated by these approaches, even if the locations of modules in the process address space were known through traditional ASLR-bypass techniques. Empowered by the *write-what-where* [26] capabilities demonstrated in Hu et al. [22], we explore the importance of server process architectures and how they affect data-oriented attacks. This connection has been critically overlooked, and we believe this oversight has dire consequences moving forward.

3.1. Defenses Against Control-Flow Hijacking

As the security community has largely acknowledged that memory corruption vulnerabilities in complex software are inevitable, defensive mitigations have most prominently targeted the control-flow hijacking steps of the exploit chain — including return-oriented programming tactics [34] and related variants. These solutions employ varied techniques to thwart attacks, such as ensuring control-flow integrity (CFI) [1, 29] or employing code diversification (*e.g.,* [5]). These approaches do not protect against *data-oriented* attacks as they are exclusively directed towards protecting the executable section of a program from being repurposed for malicious means, and do nothing to enforce the integrity of *non-control data* that is read or written by the application.

4. Goals And Adversarial Model

Given the fact that asynchronous server architectures such as Nginx handle many client connections in the same long-lived server process, our goal is to show realistic attack scenarios in which data-oriented attacks have expressive power rivaling that of control-flow hijacking exploits against web servers. Moreover, we seek to show that in some respect, data-only attacks are more attractive from an adversarial perspective than attacking control flow, since they tend to be especially covert from a system-monitoring perspective, and also obviate the need for further privilege escalation attacks once the server worker process has been exploited.

4.1. Adversarial Model

As alluded to earlier, recent work [31] has assumed the full powers of arbitrary read and write exploitation against web servers and the ability to trivially defeat ASLR given these primitives. However, our extensive research into the actual remote server exploits seen in the wild — as well as published research on the matter [22] — led us to question that assumption, and instead limit our adversarial model to one in which the adversary has the powers of *arbitrary write*, but only *linear heap disclosure*. Critically, unlike prior work, we do not assume the adversary can read data from arbitrary addresses in memory since we see no supporting evidence for this ability in real-world server exploits. Our attacks are demonstrated against Nginx, the industry leader in scalable, event-based server architectures. For simplicity, we assume the adversary has access to debug symbols, which is a realistic assumption given that the two most popular web servers[1], namely Apache and Nginx, are both open source.

5. Approach

Even under the assumption that an adversary can leak heap memory and overwrite arbitrary data in process memory, there are several hurdles that must be overcome to achieve viable data-only attacks against asynchronous web servers. First among these is identifying data that when overwritten will have the intended high-level effect of injecting malicious web content that would result in drive-by downloads or disabling services that provide privacy and confidentiality. Next, having identified this data, we must find ways to reliably overwrite it to meet the desired objective. Lastly, to fully explore the power of this threat, we seek ways to automate the steps as much as possible.

5.1. Memory Access Tracing

To address the first challenge, we provide a technique for tracing the memory accesses committed by a web server in servicing a request, and explain how these accesses can be inspected to identify data that is critical to server execution as configured by website administrators. In other words, we aim to identify data consulted on every incoming request that when overwritten will cause the server to behave differently than expected. Unexpected behaviors include serving malicious drive-by download content along with the original benign web pages, or downgrading the connection security of HTTPS without warning.

Our solution uses Intel's Pin framework [25] to record all reads directed at the `.data` section of the main executable's memory from the time the server receives an incoming HTTP request until the service of this request is complete. For each read, we also record the instruction pointer which issued the read. Next, in an offline phase, we use debug symbols to construct a timeline of data accesses made when servicing a request, including the variable name and offset in the `.data` section that was accessed as well as the function name and offset that issued the access. We trace accesses to the `.data` section (rather than the heap) because they tend to offer better insight into the high-level operations that take place while a server is processing a request. Specifically, the

1. Together, these servers account for 83% of the market share [42].

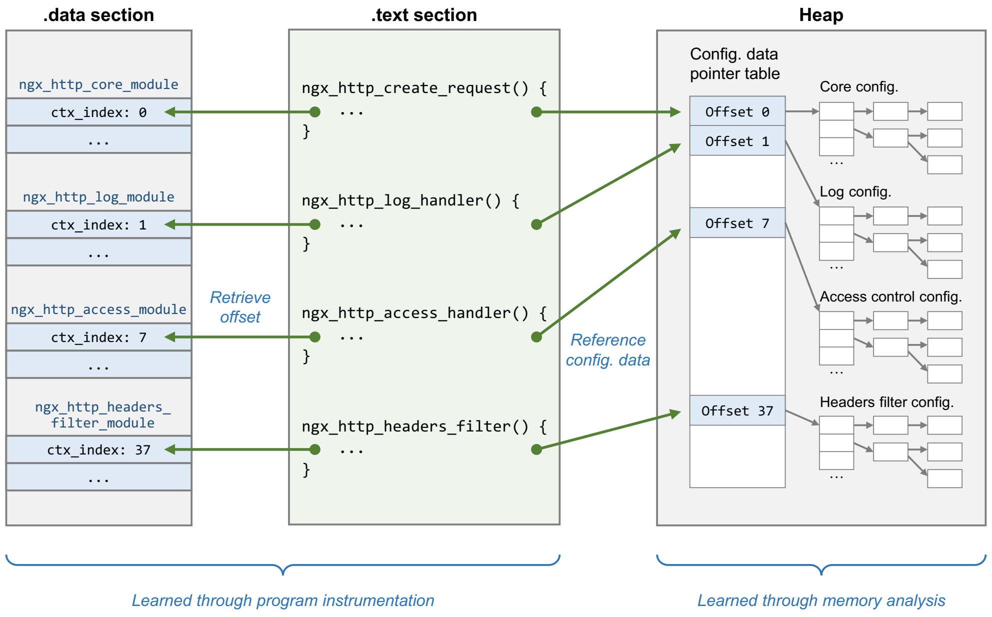


Figure 2: NGINX exploit diagram. Through program instrumentation and memory analysis, an attacker can locate the entries of interest in the configuration data pointer table, and overwrite them to point to malicious entries.

`.data` section often contains top-level pointers to complex per-module data structures which are spread throughout the heap, and this top-level is generally a good starting point for the live memory analysis techniques explained shortly. Moreover, the heap is accessed thousands of times more often during request processing than the `.data` section, and thus it is more difficult to associate high-level server operations with individual memory accesses. Lastly, even while instrumenting a program it is often difficult to associate individual allocations with the type of object that will reside at the given heap location, thus lessening the advantages provided by debug symbols.

The reader may be wondering why we do not simply conduct manual source code analysis to identify where critical configuration data is accessed in the server program. In fact, we initially took this manual approach, but soon realized that the complex nature of asynchronous web servers (in the way they chain modules and functionality together through callback functions) made for much difficulty in manually tracing the flow of execution that occurs while handling even the simplest of requests. Said another way, the performance optimization gained by asynchronous server architectures comes at the cost of code simplicity, as every small module of processing that takes place in servicing a request must be chained together through complex data structures rather than following a simple, sequential order. Such modular code design is an essential component of asynchronous web servers, as there are no thread or process objects to save the code execution state of a partially-formed response while waiting on some resource (*e.g.,* a file from disk). Instead, small code modules accomplish simple tasks that can be asynchronously invoked to perform some step towards generating a response. Thus, following the control flow and data accesses of asynchronous web servers through manual source code inspection is a difficult task, and for that reason, we resorted to program instrumentation to help identify security-critical data.

For pedagogical reasons, we note that a sample memory trace for Nginx to service an HTTP GET request contains less than 150 accesses to the data section, so it is feasible to manually identify data of interest. For example, the 96th access directed at the data section in our trace originated from `ngx_http_access_handler()`, which accesses data at offset 0 within the `ngx_http_access_module` structure. With a quick inspection, it becomes clear that the function is referencing an access control configuration data structure on the heap, using an index stored at `ngx_http_access_module` + 0 to retrieve the pointer to this data. Given such a memory trace, an adversary can easily hone in on some important access control related configuration data in memory. While this example may seem overly simple, we found that additional code paths we identified in Nginx, that consult in-memory configuration data structures for other modules (*e.g.,* SSL module, security headers module, error and access logging modules), are just as straightforward to analyze.

5.2. Corrupting Data for a Desired Effect

Armed with the ability to locate sensitive data within a program, the next challenge involves determining how to overwrite that data for the intended degradation of server security — without introducing unstable behavior to the server. In this work, we restrict our attacks to influencing the in-memory representation of configuration data. Specifically, we seek to understand how different server configuration options cause in-memory data structures to be populated with different data. This objective could conceivably be achieved through manual source-code analysis, tracking the data flow of information from configuration file to in-memory structures. However, as discussed in Section 5.1, the complex nature of callback functionality to support asynchronous server processing means such manual analysis is a non-trivial task. Alternatively, related work by Hu et al. [21] uses taint tracking to identify data in a server that is influenced by directives in a configuration file, but this is an unnecessarily complex approach that generates a vast search space,[2] and at times requires the adversary to fall back on manually specifying the security-sensitive data in an application.

Instead, we leverage the information gained from the memory tracing step to conduct live memory analysis of a running server application in order to provide intelligence on the low-level state of security-critical data structures and how they can be manipulated. Specifically, our solution for live memory analysis assumes that step one of our attack workflow has identified a spot in the code that references the given in-memory configuration data structure in question. Considering Nginx in particular, we observe that it has unique data structures representing the configuration for its different processing modules (*e.g.,* SSL module, GZIP module, access control module), and that each of these modules consult those data structures in determining how to respond to a request.

In this way, for an arbitrary server processing module, we can set a breakpoint on a location in the program that obtains a pointer to that module's configuration data, and run the server with different configuration options set, investigating how those different high-level configuration directives map onto the low level in-memory data structures once they have been populated in process memory (these in-memory data structures are shown on the right in Figure 2). With the application paused at a place where we have a reference to this process memory, we can combine debug symbols with access to raw process memory to construct an image of how the different configuration data structures for given modules are populated based on different specifications in the configuration file. In essence, we build a memory analysis framework that produces a live snapshot of a given structure, including following pointers to other structures and capturing their snapshots recursively. The left side of Figure 2 shows how our instruction tracing step helps us hone in on spots in the code that reference configuration data structures — specifically via the *config data pointer table*. Together with the results of our live memory analysis technique (shown on the right), these two frameworks help us leverage our assumptions of *linear heap disclosure* and *arbitrary write* to locate security-critical objects in memory and corrupt them for malicious effect.

The output of our memory analysis framework is a

2. A significant fraction of all data in a web server depends on the configuration file in one way or another.

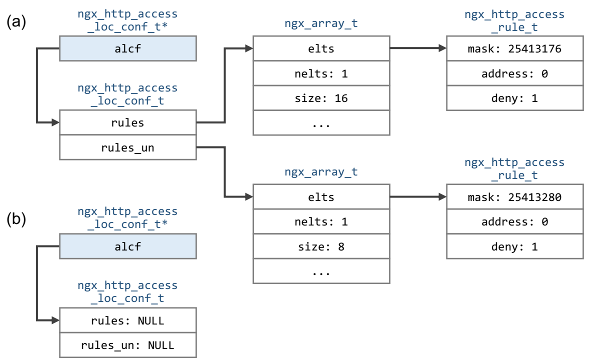


Figure 3: Extracted data structures by our memory analysis framework when configuring Nginx (a) to deny all access, and (b) to not perform any access control.

human-readable printout of the data structure as well as a copy of the data structure in binary format. There are two distinct abilities that this memory analysis approach affords the user. First, the framework can identify places in a given configuration data structure that vary for different configuration settings. Running a program multiple times with different configurations and performing a simple *diff* on the output of the memory analysis allows the user to quickly get a sense of the changes in low-level data structures that occur in response to issuing different high-level configuration directives to the application. This is useful for determining the elements in a data structure whose runtime modification will essentially *re-configure* the server, causing it to behave differently than was intended by the configuration settings. For many of Nginx's processing modules, it is a non-trivial task to hone in on which fields in the associated configuration data structure must be (recursively) altered to cause the server to operate insecurely without introducing some unexpected behavior. This is because the same configuration structures often appear very differently in process memory, depending on the directives given in the configuration file. Our framework relieves the burden of needing to understand all of these complex interdependencies in the configuration data structures, instead forcing the application to generate the different versions of the structure and making it easy to observe the differences.

Figure 3 shows example outputs of running our memory analysis framework on the access control configuration structure after having configured the server to (a) deny all access and (b) to not impose any access control (default behavior). Many of the configuration structures in Nginx are much more complex with many levels and members, but this example illustrates how the memory representation of a structure changes for different configuration directives. The differences in these structures for different configuration directives completely determine how the server responds to a given request in terms of access control. We refer to a snapshot of the data structure our framework creates as a *deep copy* of that structure since it recursively records pointers to other structures and their values.

A second benefit of our framework is the ability to extract from memory a full copy of a given data structure, outputting a *flattened* version of the arbitrarily deep multi-level structure. Implementation-wise, this involves arranging all the objects from the multiple levels of the data structure into a contiguous buffer and ensuring pointers from one level to another target the correct offsets. The ability to output this flattened structure is especially useful when considering that we assume an adversary to be lacking the ability to read arbitrary server memory. If an adversary is able to control a pointer to the top level of some multilevel configuration structure, they only need to redirect this top level pointer to a full deep copy of the configuration structure in question. The full copy is necessary since the adversary does not have the ability to *follow* pointers in the data structure to the elements they desire to modify, due to our assumption of lacking arbitrary memory disclosure capability (see §4). In this scenario, the adversary would use our framework to generate a flattened copy of some configuration structure featuring the desired insecure directives, write this buffer to server memory, and redirect the top-level pointer to reference this injected structure.

5.3. Memory Analysis Framework

Our live memory analysis framework for processing configuration-related data structures is implemented as a GDB Python plugin. With this framework in hand, one can take live *deep copy* snapshots of a given data structure and compare them across different configurations, thereby understanding how differences in configuration directives map to differences in process memory state for. Our memory analysis framework is effective in that it is generic to any arbitrary structure in the memory of a program for which debug symbols are available. However, there are a few limitations that are consequences of the C programming language, which is the source language for both Nginx and Apache. In what follows, we discuss hurdles we encountered when using our framework on Nginx. While these obstacles could be overcome through manual inspection of the source code, we present the techniques we used to overcome them without such manual effort.

Void pointers. At the first instance when our recursive memory analysis encounters a member of a struct that is of type `void*`, we will not know how to treat the structure referenced by that pointer given only debug symbols. However, there is a straightforward workaround for this issue: we can use the tracing technique (§5.1) to pause execution at a place where the structure is referenced and then set a memory access breakpoint on the location of the `void*` pointer. Upon resuming execution and triggering the breakpoint, we record the line of code associated with the current program counter and note the corresponding source code for the destination type of the cast from `void`. Thenceforth, we add a rule to the memory analysis framework to always treat a specific `void` member in a given structure as a given type for the application under inspection.

Unions. Similarly, we will not know initially how to treat a variable of type `union`; in which case a single variable can be interpreted as multiple types. This problem is an easier version of the issue with `void` pointers, and as such we use the same approach as described above.

Pointers treated as the base of an array. When an array is defined as part of a structure, we can use its statically determined size to know how many objects are contained in it, and recursively process them accordingly. On the other hand, using only debug symbols, there is no way to distinguish when a program treats a pointer type as the base of an array containing multiple items versus simply treating it as a pointer to a single object of the given type. Luckily, this most often occurs with null-terminated C-style strings of type `char*`, and thus we treat `char*` variables as arrays by default, processing memory until a null byte is encountered.[3]

Overall, our memory analysis framework vastly decreases the amount of semantic knowledge necessary for observing the runtime memory layout of security-critical data structures. While not perfect, the framework was sufficiently effective to enable a wide range of attacks against Nginx without performing manual source code analysis to reason about the structures used in the application.

6. Case Study

As shown in Figure 2, our program tracing and memory analysis frameworks enable the identification of critical configuration data in Nginx, and provide an understanding of how that data is accessed by the program. A key realization is that given the ability to control the *config data pointer table*, an adversary could trick the server into referencing any spot in process memory and interpreting it as the given type of configuration data structure. Moreover, since Nginx's asynchronous worker processes are long-lived and handle many connections, corrupting this data in a worker process will affect *all future* requests. In the case of Nginx, although this configuration data (and associated pointer table) is on the heap, there is only a single copy that is referenced throughout the lifetime of the process. Therefore, an adversary who could corrupt the pointer table and control some part of process memory could write fake configuration data structures into memory, cause entries in the pointer table to point to these fake structures, and trick the program into behaving differently from the way it was configured.

In order to accomplish this attack, the adversary must be able to (1) locate the single unique copy of the config data pointer table on the heap, (2) write a data payload somewhere in memory such that it will neither be corrupted in the future by the process nor itself corrupt any meaningful data in use by the process, and (3) create a *faux* configuration data structure containing the desired malicious parameters. Armed with these capabilities, an adversary can coax the server into behaving as desired, without ever hijacking its control flow. Worse yet, by corrupting this configuration data, the adversary can have a long-lived effect on the server, leaving behind little forensic evidence.

3. There are more complex heuristics that could be performed to predict whether a given variable points to an array, but we did not find this to be necessary for the security-critical data structures analyzed in this work. Such an exercise is left for future work.

6.1. Experimental Setup

Before describing the details of this case study for Nginx, we first relay some background experiment setup in terms of the deployment scenario that we use to evaluate the feasibility of our approach in real-world scenarios. A vitally important aspect of evaluating the behavior of a web server is the ability to interact with that server from client endpoints such that the type and frequency of client interactions are controlled for all experiments conducted. Importantly, we enable various kinds of functionality on the Nginx server which we believe is an accurate reflection of common real-world use cases. Specifically, we ensure that the server handles a mixture of HTTP and HTTPS connections, serves different types of static content (*e.g.,* HTML and JPEG files), and serves different types of dynamic content including PHP scripts. Likewise, we ensure that any time we issue client requests to the server, the requests are a diverse mixture of GET/POST requests, HTTP/HTTPS connections, requests for different types of static content, requests for URLs that exist on the server as well as some that do not (triggering an Error 404 response), and requests for different types of dynamic content. The distribution of these requests is derived from server logs from a popular campus server. Our Nginx server ran on a quad core, 8 thread, Intel i7-2600 processor with 16 GB of main memory.

Our goal in the experimental setup was to exercise many code paths on the server. This is essential for several of the experiments we run, including evaluating the feasibility of using heap disclosure to leak specific objects, as well as of finding safe areas in process memory into which we can write fake data structures. For the rest of this section, whenever we mention issuing requests that target our Nginx server, those requests are distributed according to the real-world variations above.

6.2. Locating the Config Data Pointer Table

Recall that our program instrumentation step allows us to hone in on locations in the code that retrieve pointers to configuration data from the *config data pointer table*. Investigation of the macro in Nginx that conducts this pointer retrieval shows that the location of the table is stored as part of every HTTP request structure in the program. The HTTP request structure, called `ngx_http_request_t` in the source code and referred to by `r` in the following example, is an object (allocated on the heap for each incoming connection) that gets passed along to the different processing modules in Nginx as they prepare the appropriate response. The following macro depicts how Nginx retrieves a configuration data pointer from the *config data pointer table* for a given module. This line of code represents the action that is depicted in Figure 2:

```
    #define ngx_http_get_module_loc_conf(r,
module)
(r)->loc_conf[module.ctx_index]
```

As shown, the `loc_conf` field within an `ngx_http_request_t` structure holds a pointer to the config data pointer table, and thus the ability to disclose the contents of an `ngx_http_request_t` object from the heap would allow an adversary to learn the location of the *config data pointer table*. With the location of the table known, an adversary could overwrite particular offsets (which correspond to different modules and are determined at compile-time) to point to an injected payload comprising a *faux* configuration data structure. While this will be discussed in more detail shortly, we now focus on how an adversary can reliably use a *linear* heap memory disclosure (*e.g.,*CVE-2014-0160, CVE-2014-0226, CVE-2012-1180) to leak an `ngx_http_request_t` object from the heap.

We use the Heartbleed vulnerability in our experiments to show that given a linear heap disclosure (in the case of Heartbleed, 32KB), its is realistic to assume that an adversary can disclose an `ngx_http_request_t` object from the heap with high likelihood — even though the location of heap data that is disclosed by Heartbleed is unpredictable and different every time. While this may seem odd at first blush, the chances of success are improved by the fact that one of these objects is allocated on the heap for *each* incoming request, so a server handling many requests simultaneously will have many instances of this object on the heap. Our approach involves triggering the heap disclosure in the server, followed by identifying an `ngx_http_request_t` structure within the disclosed 32KB of arbitrary heap data. To validate the right structure has been found, we perform pattern matching based on predictable data contents of the `ngx_http_request_t` structure. In the experiments that follow, we evaluated the success rate of leaking the desired structure on a moderately loaded server averaging 25 connections per second — a number derived from data we collected for one of the main web servers on our campus.

If no other clients are interacting with the server at the time a disclosure is performed, there may be only a few `ngx_http_request_t` objects on the process heap. However, adversaries can increase their chances of success by preparing the process heap with innocuous HTTP requests before performing the disclosure. It is important that these requests are performed in parallel, so that multiple `ngx_http_request_t` objects are allocated on the heap for the different requests. Therefore, in each run of the experiment, we allow the adversary to prime the server by first issuing $n \in 0 \ldots 30$ simultaneous HTTP requests before performing a disclosure. That exercise is repeated for $d = 50$ disclosure attempts for each value of n, and to ensure there are no lingering side effects, the server is restarted before proceeding to the next value of n.

Figure 4 (top) shows the average success rate of finding an `ngx_http_request_t` structure after prepping the server and subsequently performing a disclosure. Notice that without prepping the server, we achieve an average success rate of 12.4%. That success rate peaks to just under 33.7% at $n = 7$ innocuous requests, before stabilizing. We believe the fluctuations after around $n = 7$ are due to intricacies of how Nginx handles connections. The attacker can increase her success rate — though at the risk of raising suspicion server-side — by instead prepping the server with requests targeted

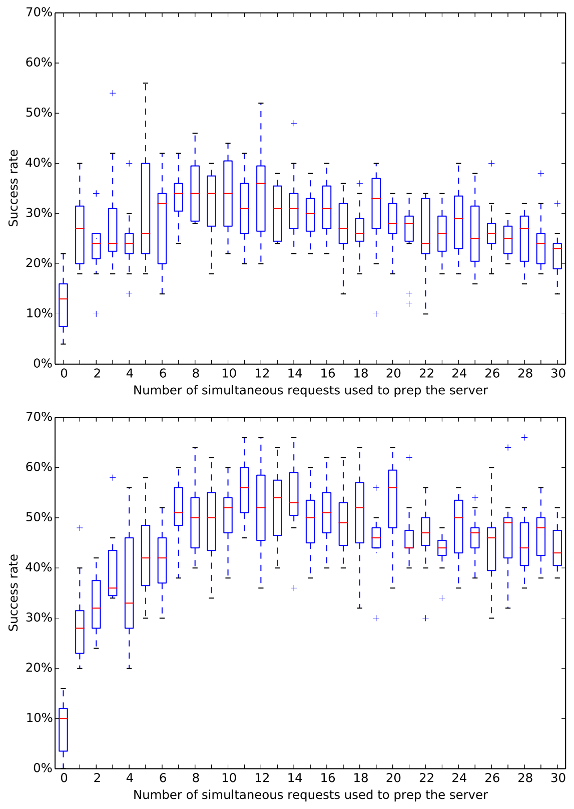


Figure 4: Disclosure success (a) for increased stealth, with adversarial prepping following the distribution of request types for varied content, and (b) when prepping targets only a PHP script. Performing 10 disclosures at a 25% individual success rate gives an overall likelihood of greater than 94%.

at a specific server-side PHP script, for example, that ties up resources slightly longer than for simply returning a static HTML page (*i.e.,* $\approx 250ms$ versus $50ms$). Figure 4 (bottom) shows that in this case, the success rate improves dramatically. In any event, our disclosure technique does not need to have a 100% success rate, since triggering the Heartbleed leak does not crash the server worker process and thus can simply be exercised multiple times until the structure is successfully disclosed.[4] Even placing a conservative estimate of the individual disclosure success rate at 25%, performing 10 disclosures would raise the overall likelihood of success to above 94%.

Lastly, we explored what happens on a more heavily loaded server scenario by issuing numerous requests from many different physical machines until a threshold is reached where our Nginx server is operating at max capacity (servicing around 130 requests per second). While maintaining this max throughput threshold on the server, we triggered d disclosures at random periods and examined the success rate for finding the `ngx_http_request_t` object. We verified that we attained and maintained a threshold of heavy use on the server by monitoring the connection logs with Splunk [40]. Even then, the observed success rate (not shown) at $n = 7$ was 12% with no prepping, 16% when the innocuous HTTP requests target an HTML file, and 32% for the PHP target.

4. Even with a linear memory disclosure vulnerability that *does* cause a crash, this does not present much of an issue as Nginx worker processes are restarted automatically after a crash.

6.3. Writing Faux Data Structures

After leaking an `ngx_http_request_t` object and determining the location of the config data pointer table, an attacker can use an arbitrary write vulnerability to *reconfigure* the server by overwriting an offset in this table and redirecting the program to accessing a fake configuration data structure. Recall that the need for creating an entire fake data structure (rather than simply overwriting elements in an existing structure) revolves around the fact that our heap disclosure and ability to find the `ngx_http_request_t` object only allows for knowing the base location of the config data pointer table. Since we do not assume the ability to read arbitrary process memory in our attacks, we are restricted to overwriting offsets in this table without being able to read the pointers that exist at given offsets in this table. Therefore, our only choice is to write an entire fake copy of a given configuration data structure to memory and redirect a pointer in the table to this location.

Having obtained the data structure format that corresponds to some unsecure configuration of a given module, we need to figure out how to write this fake structure into a safe place that will not disrupt the execution of the server. Equally importantly, we do not want the server to corrupt our fake data structures at any time in the future. Although it may be possible to write these fake data structures to the process stack or heap, in our approach we elect to write our payload into an unused portion of the data section. This location is attractive because (*i*) knowing the base address of the data section allows an adversary to have full knowledge of the offsets (determined at compile time) of different variables and structures in it, and certain swaths of memory in the data section may never be used by the worker process in the Nginx model; (*ii*) the size of the data section does not dynamically change, unlike the process stack/heap; (*iii*) in general, any part of the stack/heap that is allocated by the worker process will be used/reclaimed at some point, which may present challenges for persistence of the written data without introducing incorrect behavior to the server.

For two key reasons, the attack techniques we demonstrate hinge on the ability to overcome ASLR. First, we need to know the absolute address of the `.data` section to determine where to write our *faux* data structures inside it. Second, we need to fix up any pointers (*e.g.,* references to function addresses) in the *faux* data structures we generate to point to the correct offsets in the given module. Beyond defeating ASLR, we must also identify offsets *within* the `.data` section that can be effectively used as *scratch space* to write our data structure payloads without worrying about touching data that is actually used by the application.

It turns out that the first requirement has a simple solution. The `ngx_http_request_t` object leaked from the heap contains multiple function pointers to predictable offsets. Specifically, its `read_event_handler`, `write_event_handler`, and `log_handler` members predictably point to three respective functions in the main executable. Critically, ASLR moves around modules in such a way that knowing the absolute address of the text section of the main executable gives the absolute address of the data section as well. In this way, our heap disclosure also allows us to compute the base address of the data section in the main executable.

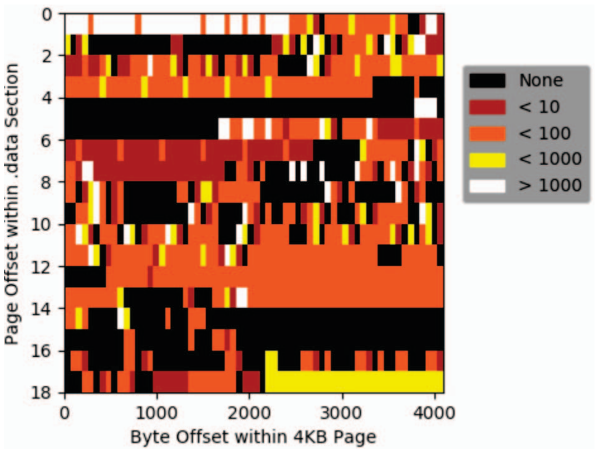
To identify offsets in the data section that fulfill our second requirement, we propose a strategy for tracing memory accesses committed by the parent and worker processes during server execution. A motivating realization for this approach is that parts of the data section are likely only used by the parent process in Nginx, and therefore after the worker process is forked to handle connections, these zones can be freely written by the worker and will never be accessed by normal program execution in the worker. Moreover, there are likely zones which are never accessed by either the worker or the parent (*e.g.,* static error pages in memory for errors that will never be triggered), which also implies the adversary can safely write to these regions.

To trace memory accesses to the data section, we instrument the server to record accesses in both the parent and worker processes from the time the parent process is started to when both processes are terminated when the server is shut down. Figure 5 shows accesses to the data section by the parent and worker processes, respectively. The heatmaps correspond to starting the server, handling 10,000 HTTP requests, then shutting down. Dark regions represent no access, while white regions denote areas that are heavily accessed.

The plots show that the parent process accesses the data section more extensively than the worker, so a remote attacker exploiting the worker process has many options for places within the data section to write their fake data structures. Moreover, the predictable code paths of the worker process mean that the same offsets are accessed over and over, so the adversary can be confident large swaths of memory will not be touched by the worker. There are even places in the data section that are not touched at all by either process (*e.g.,* in pages 1, 4, 5, 14, 15). This is due to the fact that some data compiled into the server (*e.g.,* static strings representing canned error page responses for unusual errors) go un-accessed even for long running instances.

6.4. Creating Valid Faux Data Structures

Next, we discuss how an adversary can construct fake data structures and write them into process memory in such a way that they will be semantically valid fake versions of the corresponding configuration data structures. To highlight the ease with which this can be done, we built our automated memory analysis framework with the capability of outputting configuration structures in a semantically valid binary format. This way, in an offline step, an adversary can



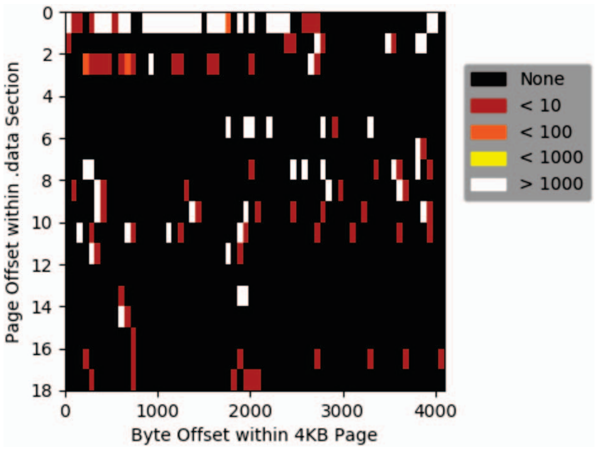


Figure 5: Heatmap showing (a) accesses in the parent process, and (b) accesses in the worker process

set the configuration for Nginx to some insecure setting, analyze the configuration data structure containing that insecure setting with our memory analysis tool, and obtain the binary format of that structure which can be written into process memory during an exploit. To successfully use the framework, the adversary must provide the absolute virtual address offset where the binary data structure payload will be written in memory. Also, if any pointers in the *faux* structure need to reference a given module (*e.g.,* function pointers in to the `.text` section of the executable), the adversary must provide the offset of this module. In particular, for the attacks we demonstrate, some of the *faux* structures that our framework successfully generates (shown in Table 1) must contain function pointers into the main Nginx executable. We provide this through disclosure of the `ngx_http_request_t` structure as discussed in Section 6.2, which contains function pointers that allow us to compute the location of the main Nginx executable.

6.5. Findings

The end-to-end exploits we performed that aptly demonstrate the power of the attacks are listed below. These exploits were performed against a running Nginx instance vulnerable to Heartbleed and a simulated remotely exploitable arbitrary 8-byte write vulnerability.[5]

1) Reconfigure the server to cease logging connections.
2) Re-enable logging on the server (useful for achieving stealthiness after exploitation is complete).
3) Reconfigure the server to use a higher *error alert level*, in essence causing the server to cease reporting anything but the most extreme errors.
4) Reconfigure the server to use the document root path / rather than the default path, allowing for leaks from the file system, including the server's private RSA key.
5) Restore normal configuration after attacks 3 and 4.
6) Control what headers are appended to HTTP responses by the server (*e.g.,* causing the server to omit security critical headers such as HSTS, X-XSS-Protection, X-Frame-Options, Referrer Policy) to disastrous effect [17, 35].
7) Enable or disable access control on the server.
8) Change the maximum SSL protocol version that will be supported by the server (*e.g.,* limiting the server to use TLS 1.0 or SSLv3).[6]

In the case of web-based malware distribution, the ability to enable access control in Nginx turns out to be especially powerful. Since the default *Error 403* page served by Nginx is stored at a pre-determined compile-time location in the data section, an adversary can overwrite elements of this simple HTML page with a custom page containing malicious web content (*e.g.,* a JavaScript exploit within a hidden frame). Then, by *re-configuring* access control on the server to *deny* access to all clients (or particular IP addresses), the adversary can force the custom *Error 403* page to be distributed by the server en masse. This capability would be a springboard for adversaries to gain widespread distribution of web malware or perform targeted attacks against a web service. Notice that with logging temporarily disabled during the attack, server-side monitors that operate off the error or access logs will not notice the attack, thereby making it extremely difficult for network operators to detect or diagnose[7] the malfeasance.

Without a doubt, these attacks demonstrate the serious threat of non-control-data oriented attacks against asynchronous web servers. Table 1 shows the sizes of the configuration data structures that were written into memory for the various exploitation scenarios. For all cases but the SSL configuration data structure, our memory analysis tool was able to automatically produce a fake configuration data structure that is semantically acceptable in order to *re-configure* the server without introducing unexpected behavior. The difficulties posed by the particular SSL configuration structure are due to limitations of the current implementation of our memory analysis framework (see §5.3).

Critically, we note that both a) *disabling server logging* and b) *disabling all security headers* can be done with a *single* top level pointer overwrite in the *config data pointer table* and do not require generating a fake structure at all. This is because each of these data structures contain a variable in their top level that when assigned a specific value causes the server to completely forgo using the associated module to process a request. Thus, redirecting the associated entry in the *config data pointer table* to point to any memory in the `.data` section which contains the given value (zero in case a, true (non-zero) in case b) at the given offset is semantically just as effective as writing the whole *faux* data structure to memory — since in both of these cases the certain value in a single variable is all that is necessary for achieving the desired *re-configuration*. We verified that this optimization works in practice. This saves the adversary a few bytes-worth of memory overwrites and simplifies the attack payload as much as possible for these powerful attacks. Importantly for case a, this means that only a *single pointer overwrite to the given offset in the table is sufficient for completely disabling the access logs in Nginx*. Thus an adversary could do this as a first step and then proceed to perform any number of connections required in order to write *faux* data structures to memory for *re-configuring* other processing modules, all while evading detection by server monitoring mechanisms.

The rightmost column of Table 1 shows the number of connections required to write the other *faux* data structures to memory once logging has been disabled. Assuming an 8 byte overwrite per HTTP request and 100 requests per keepalive connection (default on Nginx), we can overwrite 800 bytes per connection.[8] This is an important consideration in the context of network traffic monitoring systems which seek to detect anomalous connection behavior. We note that even in a less-ideal situation where the specific vulnerability requires multiple requests to trigger the *arbitrary write* or only affords an overwrite of lesser size, the approach could still be extended to evade detection as even if we increase the number of connections required by an order of magnitude, the attack would likely go undetected on a busy server (*e.g.,* twitter handled 200–300 connections per second, on average, in 2009).[9]

6.6. Empirical Analysis

To assess the potential impact of attacks of the kind disclosed herein, we performed an empirical evaluation using data provided by a cloud-based service, called Censys [15]. Censys maintains an up-to-date snapshot of the hosts and services running across the public IPv4 address space. Starting in August 2015, Censys routinely scans the public address space across a range of ports and protocols, and

5. The realism of this threat model in real-world deployment scenarios is discussed throughout this work, including in Sections 2, 3, 4 and 6.6.

6. TLS 1.0 is supported by all major browsers and even the insecure SSLv3 was supported in recent browser versions, including Safari for OS X 10.10 and iOS 8 [44].

7. For example, Cloudflare's analysts relied almost exclusively on server logs to understand what might have been leaked. See https://blog.cloudflare.com/quantifying-the-impact-of-cloudbleed/.

8. Per Hu et al. [22], CVE-2013-2028 can be used to accomplish this arbitrary write, in addition to leveraging the Heartbleed bug for a *linear memory disclosure.*

9. See http://highscalability.com/scaling-twitter-making-twitter-10000-percent-faster.

TABLE 1: Size of data structures for different configurations.

Structure	Initial State	Initial State Struct Size (Bytes)	New State	New State Struct Size (Bytes)	Automatic Generation Successful?	No. Conns.
Logs Config	Normal Logging	802	No Logging	40[1]	Yes	1
Logs Config	No Logging	40[1]	Normal Logging	794	Yes	1
Core Config	Default Error Alert Level	1417	Elevated Error Alert Level	1417	Yes	2
Core Config	Default Document Root Path	1417	Document Root Path: /	1397	Yes	2
Headers Config	No Headers	32	Use Security Headers[2]	534	Yes	1
Headers Config	Use Security Headers[2]	534	No Headers	32	Yes	1
SSL Config	Use up to TLS 1.2	12615+	Use up to TLS 1.0	12615+	No	16+
SSL Config	Use up to TLS 1.0	12615+	Use up to TLS 1.2	12615+	No	16+
Access Ctrl Config	No Access Control	16	Deny All	112	Yes	1
Access Ctrl Config	Deny All	112	No Access Control	16	Yes	1

[1] See Section 6.4 on how creating a *faux* structure is not necessary here. [2] HSTS, XSS-Protection, X-Frame-Options and Referrer Policy.

validates the resulting data via application-layer handshakes. The framework also dissects the handshakes to produce structured data about each host and protocol. We use data from Censys to examine the number of hosts that were vulnerable to Heartbleed (CVE 2014-0226) or were running versions 1.39 or 1.40 of Nginx that were affected[10] by CVE-2013-2028. We examined data for the earliest day (*i.e.,* 7/8/2015) for which Censys provides scans for Heartbleed and port 80 scans for the IPv4 address space.

The results are quite troubling — even 16 months after the initial disclosure on April 7, 2014 [14], 255,161 servers were still vulnerable to Heartbleed, and 3599 servers were running vulnerable versions of Nginx. This is quite disheartening given that there were no less than five *major* releases of Nginx after version 1.4 and before the snapshot date, yet still several major websites were running a significantly outdated version. While only 75 network objects (*i.e.,* 2 domains in the Alexa's top 1 million on 7/8/2015 and 73 IPs) were potentially vulnerable on the day of the Censys scan to *both* of the CVEs relied upon in this paper, the results would certainly have been far worse closer to ground zero. The fact that there are only limited automatic updates for web servers (unlike the browser market), coupled with the observation that many servers may go unattended for long periods once deployed, may be contributing factors to why these servers went unpatched for so long.

To understand how many clients may have been exposed to these potentially vulnerable servers, we used a large passive DNS [43] datastore to analyze 6 days worth of DNS lookups in May 2017. We only analyzed the subset of 3133 vulnerable servers that were in the Alexa Top 1 million on 7/8/2015. Figure 6 shows the observed DNS resolutions attempted by clients to these network objects during the monitored period. We observed 481,122,464 resolution attempts from 5,607,805 clients to servers that were subject to either vulnerability. The lookup volume to the 75 network objects with both vulnerabilities was far less — only 19 on average, but several of these servers are now defunct. We note that our statistics are lower bounds on what the

10. Note that from the Censys data it is impossible to tell where the sites were running patched versions and so the numbers reported here could be an over-estimate.

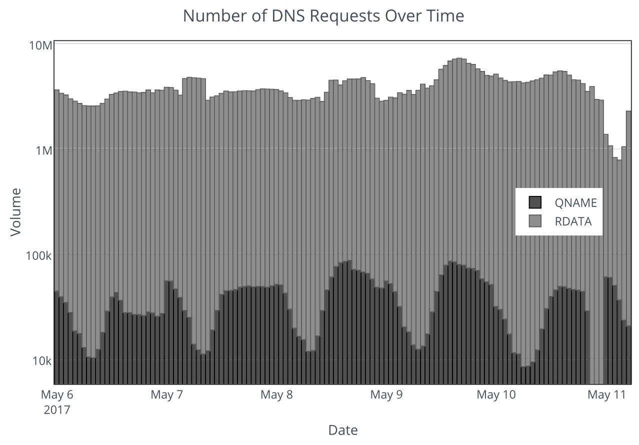


Figure 6: DNS resolutions

potential affected client population would have been like on 7/8/2015 because we are effectively sampling as the passive DNS data is from the vantage point of a single provider in the US, and several of those domains are popular in regions outside our purview.[11]

6.7. On the Assumption of Arbitrary Write Capabilities in Multi-Core Scenarios

Recall that on an 8-core system, for example, Nginx starts one main parent process which then spawns 8 different long-running worker processes. This might seem a problem for an adversary when performing multiple *arbitrary writes* to worker process memory, as the writes may be spread across multiple processes, thereby disrupting the attack. Yet, the adversary can easily sidestep this potential issue by taking advantage of the HTTP connection keepalive functionality. Specifically, a given keepalive connection will always reside in the same Nginx worker process for the lifetime of that connection. Additionally, we found that all the security-critical configuration data structures are instantiated on the heap by the parent process as part of server start-up (*i.e.,* before the *fork()* calls that spawn the worker processes),

11. Therefore, lookups for domain names that, e.g., may be popular in Asia or Europe, will not be well represented in the estimates we provide.

so all the worker processes inherit the same address space containing the structures (*e.g.,* the *config data pointer table*) at the same addresses. Thus, even in a multicore setting, a disclosure that leaks the address of a structure in one worker is sufficient for knowing the location in all processes.

6.8. Applicability To Other Modern Web Servers

As a step toward assessing the generalizability of our techniques, we applied a similar analysis to another web server that also supports processing simultaneous connections — namely, Apache. Specifically, using our program tracing and memory analysis frameworks (§5), we investigated whether the key architectural weaknesses we brought to light earlier are also central to the way Apache processes connections.

We remind the reader that the classic processing model employed by Apache provides isolation between clients and is less vulnerable to memory corruption attacks that trigger bugs in one connection to affect the processing of a different connection. However, Apache no longer runs in the classic mode by *default*, preferring to employ thread-based connection processing, in which many different connections share global data that is not specific to a given thread. Thus, to gain insight into the susceptibility of Apache, we analyzed its multithreaded "event" and multi-process "prefork" worker models.

Given that an end-to-end proof of concept against Apache would be beyond the scope of this paper, we focused our cursory analysis on answering two questions that we believe are key to understanding the susceptibility of Apache servers, specifically: (*i*) does Apache store a single copy of its global configuration data in such a way that corruption of this data affects how all threads in the process service their respective connections? (*ii*) are there readily accessible data structures on the heap that point to such global data, such that a linear heap disclosure could reliably identify the location of the configuration data? In short, the answer to both of these questions is yes.

In the same fashion as was done for Nginx, we used our program instrumentation workflow to identify global configuration data structures in Apache that are referenced by each thread during the processing of client connections. We supplemented our analysis with a review of the source code and found that Apache stores the server configuration in a global data structure `server_rec` on the heap. The configuration-related data is initialized during server startup by the control process (routine `init_server_config()` in `server/config.c`)), and resides in the memory of the child processes after forking.

Among the basic configuration fields, we found that the server configuration contains a `module_config` vector that stores pointers to configuration data structures of all enabled modules. Apache relies on a set of macros operating on the vector `module_config` to obtain the configuration of each registered module. This pattern closely resembles the module configuration code in Nginx. Instrumentation of the web server process using our program tracing and memory analysis framework indicated multiple accesses to the server configuration struct, confirming

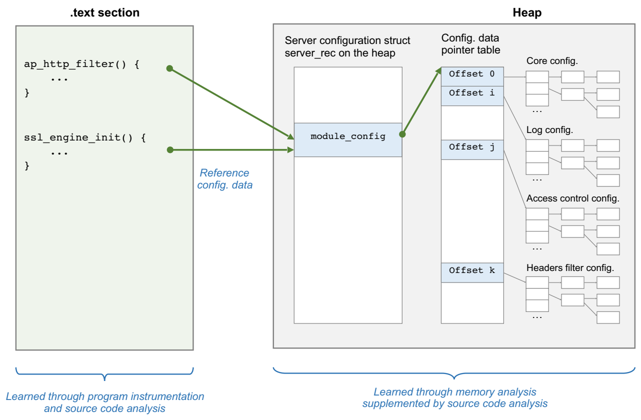

.text section
Heap
ap_http_filter() {
...
}
ssl_engine_init() {
...
}
Reference config. data
Server configuration struct server_rec on the heap
module_config
Config. data pointer table
Offset 0
Offset i
Offset j
Offset k
Core config.
Log config.
Access control config.
Headers filter config.
Learned through program instrumentation and source code analysis
Learned through memory analysis supplemented by source code analysis

Figure 7: Connected structures in Apache.

our suspicion that modifying the configuration structure on the heap will affect the processing of all subsequent connections for a given process. As a specific example, notice that the function `ap_http_filter()` accesses the global `server_rec` structure when deciding how to respond to a GET request (*e.g.,* `core_server_config *conf = (core_server_config *) ap_get_module_config(f->r->server-> module_config, &core_module))`.

Digging deeper, analysis of the `httpd` source code revealed that two data structures (`conn_rec` and `request_rec` representing HTTP connection and HTTP request, respectively, contain pointers to the global `server_rec` data structure. Given the observation that each HTTP connection will result in a `conn_rec` structure, and possibly multiple `request_rec` structures are allocated on the heap, we believe that leaking a pointer to the global server configuration should be as viable as in the Nginx case. Successfully exploiting this at runtime is left as an exercise for future work. Conceptually, after locating the `server_rec` structure containing the server configuration, the attack would simply proceed as in the Nginx case, *i.e.,* one needs to create faux data structures and find a suitable place to write them in the process memory.

These findings suggest that the framework provided in this paper can be extremely helpful in diagnosing security weaknesses in modern servers. In summary, our analysis of two major asynchronous web servers lead to similar findings: performance optimizations that drive the architectural design decisions in these applications significantly amplify their susceptibility to data-oriented attacks.

7. Mitigations

Enforcing full memory safety to unsafe languages can essentially block all memory corruption exploits. Unfortunately, this entails both spatial and temporal safety, which results in a prohibitively high cost. Indicatively, when CETS [28] is coupled with SoftBound [27] to achieve full memory safety, the resulting approach incurs an average overhead of 116% for the SPEC benchmarks [28]. Even when focusing on spatial safety alone, runtime overheads are considerable. By trading some extra memory for performance, baggy bounds checking [3] is currently one of the most efficient object-based bounds checking approaches, although its performance overhead is still prohibitively high, at an average of 60% for the SPEC benchmarks.

Thankfully, although the impact of web server process models on securing against memory corruption attacks has been largely overlooked, there has been a renewed interest in techniques that seek to thwart data-oriented attacks in general [6, 7, 8, 9, 13, 16, 32, 37, 38]. These defenses attempt to ensure that the *data flow* of a program follows paths intended by the programmer. To see why that is important, recall that a key aspect of the recent data-oriented attacks is the ability to launch an exploit by corrupting heap memory. For the most part, the defenses proposed to counter such attacks seek to enforce the integrity of data flow in a program by assuring that memory references only access data in the manner intended by the programmer. For instance, several defenses have been proposed based on source-compatible solutions [2, 6, 8, 37] that require no assistance from the programmer. At a high level, these approaches instrument data accesses (*e.g.,* using compiler frameworks like Phoenix[12]) combined with pointer analysis techniques [19, 20] to determine whether a given data access should be allowed at runtime. Alternatively, other approaches [7, 10, 32] leverage programmer assistance to identify security critical data, after which a multitude of strategies for hardening the program against corruption or leakage of that data are deployed. We discuss each in turn.

Data Flow Integrity Through Instrumentation (without programmer assistance)

Most contemporary defenses against data only attacks rely on pointer (points-to) analysis [36] to ensure the integrity of data flows in an application. For example, KENALI [37] uses an automated approach for identifying security critical data paths and sequestering them in their own address space, which is then protected by a data flow integrity solution [2]. The goal of pointer analysis is to compute an approximation of the set of program objects that a pointer variable or expression can refer to. Although pointer analysis is (in general) an undecidable problem, there are heuristics for approximating which pointers point to what objects [19, 20, 36]. While these approximation algorithms are generally thought of as best effort compiler techniques for eliminating dead code and identifying programmer errors, their use for enforcing data flow integrity was nonetheless popularized by Castro et al. [8] and Akritidis et al. [2]. The idea is that given the list of objects that each pointer in a program can access, it should be possible to instrument programs to ensure at runtime that memory objects are only accessed through pointers that are allowed to reference the given memory. Although effective in certain scenarios [2, 6, 8, 37], pointer analysis is not a holistic approach for enforcing data flow integrity, due to the fact that the algorithms are ineffective in many scenarios and do not yet handle the complexities of real-world software [36].

12. See https://en.wikipedia.org/wiki/Phoenix_(compiler_framework)

As a case in point, WIT [2] uses points-to analysis to compute the set of objects that can be modified by each instruction in the program. Given that pointer analysis is only an approximation algorithm and cannot provide strong security guarantees on its own, WIT supplements pointer analysis with software guards between objects that prevent overflows from corrupting adjacent objects. Still, these guards are not supported in the heap, which is left vulnerable. Importantly, Akritidis et al. [2] note that WIT should be capable of preventing attacks that violate write integrity, but the number of attacks that violate this property depends on the precision of the points-to analysis. Similarly, the approach of Bhatkar and Sekar [6] hinges on the accuracy of pointer analysis in order to provide any security assurance. The idea of that work is to associate a mask with each memory object in a program, so that in order to reference memory correctly, a code path must be instrumented to first unmask the memory before using it in an operation. In circumstances where pointer analysis is not effective, the approach of Bhatkar and Sekar [6] must resort to sharing the same mask between many objects.

In a related effort, Song et al. [37, 38] suggest approaches for protecting security-critical data in operating system kernels. Essentially, they propose an automated approach for locating security-critical data in memory as well as a solution for isolating the data by means of a shadow address space and context switching at runtime. Unfortunately, as their approach builds upon techniques like WIT [2] to enforce data flow integrity in the protected shadow context, it suffers from significant drawbacks when dealing with dynamic memory allocations.

Protection of Specific Critical Objects (as specified by the programmer)

The second category of mitigations against data-oriented attacks includes approaches that steer clear of the problems imposed by complex pointer analysis approximation algorithms. That is, rather than enforcing fine-grained data flow integrity, defenses in this category separate sensitive (as denoted by the programmer) and non-sensitive objects into two regions, and ensure that data does not flow between regions or between objects in the sensitive region [7]. For ease of annotation and data flow tracking, data structures are often labelled with the same sensitivity as their sub-objects, and implicit sensitivity is applied by the compiler to objects that interact with sensitive objects.

To determine the taint propagation of object sensitivity, all the explicitly and implicitly sensitive variables are found at compile time using inter-procedural and field insensitive data-flow analysis. Given the hardness of such data-flow tracking, most proposed algorithms are forced to be conservative in their approximations to avoid crashing the program if two objects that interact at runtime were labeled with different sensitivity levels at compile time. Consequently, the amount of data marked as sensitive in an application can easily blow up, even if the programmer only marks a single object as sensitive. Hence, such defenses simply reduce to memory safety policies like SoftBound [27] that performs bounds checking on memory accesses to objects, and as such, are ineffective in honing in on the truly critical security data of the application, or protecting just the subset of memory as originally specified by the programmer.

Container and Microservices Architectures

Recently, there has been tremendous interest in microservices (*i.e.,* architectural patterns in which complex applications are composed of small, independent processes that communicate with each other in a secure manner). Indeed, there are now academic conferences with sessions focused almost exclusively on best software engineering practices for microservices (*e.g.,* The Software Architecture Conference).

One direction could be to follow the lead taken by modern browser designs for providing process isolation [12]. Indeed, although the browser security community has learned to heavily rely on code refactoring, sandboxing, and multiprocess architectures to protect its users from attacks, to date the process architectures for web servers seem to have only considered performance and robustness, but not security. That said, even for the browser community where security has been a longtime concern, data-only attacks still pose a daunting threat and have been recently used to disclose sensitive data from a victim domain that resides in the same process as the attacker domain [23, 30]. Nevertheless, although the right balance is difficult to achieve in practice, the landscape for defenses has not been well explored and is an area ripe for research. We hope our findings will stimulate further research in that direction.

8. Conclusion

Taken as a whole, our instruction tracing method and live memory analysis framework demonstrate the ease with which an adversary can perform powerful attacks against asynchronous web servers that service many clients in the same process. We demonstrate how the control-flow hijacking and privilege escalation steps in the web server exploit chain can be circumvented to significantly increase the realism of using memory corruption attacks to subvert these critical systems. Moreover, as the rest of the server industry has been trying to keep up with the impressive scalability provided by Nginx through its asynchronous architecture, Apache and other competing server solutions are refactoring themselves to be more aligned with the model of handling many different client requests within the same server process. This drive in server architectures towards scalability and away from memory isolation between requests opens the door for the feasibility of non-control data attacks against web servers that were previously not vulnerable to such attacks in their classic architecture. As the increasing majority of the World Wide Web's most trafficked server side applications share critical data between many mutually distrusting clients, we expect this issue to only become more prominent going forward.

9. Acknowledgments

We express our gratitude to our shepherd, Robert N. M. Watson, and the anonymous reviewers for their suggestions on how to improve the paper. We also thank Murray Anderegg and Chaz Lever for their efforts in deploying the infrastructure used in this study, and their assistance with the collection and analysis of DNS data. This work is supported in part by the Department of Defense (DoD) under awards FA8750-16-C-0199 and FA8750-17-C-0016, as well as the Office of Naval Research (ONR) under awards N00014-15-1-2378 and N00014-17-1-2891. Any opinions, findings, and conclusions expressed herein are those of the authors and do not necessarily reflect the views of the DoD or ONR.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-Flow Integrity. In *ACM CCS*, 2005.

[2] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing Memory Error Exploits with WIT. In *IEEE Security & Privacy*, pages 263–277, 2008.

[3] P. Akritidis, M. Costa, M. Castro, and S. Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX Security*, pages 51–66, 2009.

[4] Apache. Core features and multi-processing modules, 2017. URL https://httpd.apache.org/docs/2.4/mod/.

[5] E. Bhatkar, D. C. Duvarney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *USENIX Security*, pages 105–120, 2003.

[6] S. Bhatkar and R. Sekar. Data space randomization. In *Detection of Intrusions, Malware and Vulnerability Assessment*, 2008.

[7] S. A. Carr and M. Payer. Datashield: Configurable data confidentiality and integrity. In *ACM Asia CCS*, 2017.

[8] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *USENIX OSDI*, 2006.

[9] Q. Chen, A. M. Azab, G. Ganesh, and P. Ning. Privwatcher: Non-bypassable monitoring and protection of process credentials from memory corruption attacks. In *ACM Asia CCS*, 2017.

[10] Q. Chen, A. M. Azab, G. Ganesh, and P. Ning. Privwatcher: Non-bypassable monitoring and protection of process credentials from memory corruption attacks. In *ACM CCS*, pages 167–178, 2017.

[11] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *USENIX Security*, 2005.

[12] Chrome Team. Site isolation summit:. Overview Videos, 2015.

[13] I. Diez-Franco and I. Santos. Data is flowing in the wind: A review of data-flow integrity methods to overcome non-control-data attacks. In *Complex, Intelligent, and Software Intensive Systems*, 2016.

[14] Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer, and V. Paxson. The matter of heartbleed. In *ACM IMC*, pages 475–488, 2014.

[15] Z. Durumeric, D. Adrian, A. Mirian, M. Bailey, and J. A. Halderman. A search engine backed by Internet-wide scanning. In *ACM CCS*, 2015.

[16] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *USENIX OSDI*, pages 75–88, 2006.

[17] S. Fogie, J. Grossman, R. Hansen, A. Rager, and P. Petkov. *XSS Attacks: Cross Site Scripting Exploits and Defense*. Syngress Publishing, 2007.

[18] J. Graham-Cumming. Incident report on memory leak caused by cloudflare parser bug, Feb 2017. URL https://blog.cloudflare.com/incident-report-on-memory-leak-caused-by-cloudflare-parser-bug/.

[19] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *ACM Workshop on Program Analysis for Software Tools and Engineering*, pages 54–61, 2001.

[20] M. Hind and A. Pioli. Which Pointer Analysis Should I Use? *SIGSOFT Softw. Eng. Notes*, 25(5):113–123, Aug. 2000.

[21] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang. Automatic generation of data-oriented exploits. In *USENIX Security*, pages 177–192, 2015.

[22] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *IEEE Security & Privacy*, 2016.

[23] Y. Jia, Z. L. Chua, H. Hu, S. Chen, P. Saxena, and Z. Liang. The "web/local" boundary is fuzzy: A security study of chrome's process-based sandboxing. In *ACM CCS*, pages 791–804, 2016.

[24] D. Kegel. The c10k problem, 2014. URL http://www.kegel.com/c10k.html.

[25] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM PLDI*, pages 190–200, 2005.

[26] MITRE. CWE-123: Write-What-Where Condition. Available from MITRE, CWE-123: Write-what-where Condition, 2017. URL https://cwe.mitre.org/data/definitions/123.html.

[27] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Softbound: Highly compatible and complete spatial memory safety for C. In *ACM PLDI*, pages 245–258, 2009.

[28] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. CETS: Compiler enforced temporal safety for C. In *Symposium on Memory Management*, pages 31–40, 2010.

[29] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In *USENIX Security*, pages 447–462, 2013.

[30] R. Rogowski, M. Morton, F. Li, K. Z. Snow, M. Polychronakis, and F. Monrose. Revisiting browser security in the modern era: New data-only attacks and defenses. In *IEEE European*

[38] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and

Symposium on Security and Privacy, 2017.

[31] R. Rudd, R. Skowyra, D. Bigelow, V. Dedhia, T. Hobson, S. Crane, C. Liebchen, P. Larsen, L. Davi, M. Franz, A.-R. Sadeghi, and H. Okhravi. Address oblivious code reuse: On the effectiveness of leakage resilient diversity. In *ISOC NDSS*, 2017.

[32] C. Schlesinger, K. Pattabiraman, N. Swamy, D. Walker, and B. Zorn. Modular protections against non-control data attacks. In *IEEE Computer Security Foundations Symposium*, 2011.

[33] F. J. Serna. The info leak era on software exploitation. In *Black Hat USA*, 2012.

[34] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *ACM CCS*, pages 552–561, 2007.

[35] S. Sivakorn, I. Polakis, and A. D. Keromytis. The cracked cookie jar: HTTP cookie hijacking and the exposure of private information. In *IEEE Security & Privacy*, pages 724–742, 2016.

[36] Y. Smaragdakis and G. Balatsouras. Pointer analysis. *Found. Trends Program. Lang.*, 2(1):1–69, Apr. 2015.

[37] C. Song, B. Lee, K. Lu, W. Harris, T. Kim, and W. Lee. Enforcing kernel security invariants with data flow integrity. In *ISOC NDSS*, 2016.

Y. Paek. HDFI: Hardware-assisted data-flow isolation. In *IEEE Security & Privacy*, 2016.

[39] R. Soni. *Nginx: from beginner to pro*. Apress, 2016.

[40] Splunk. Operational intelligence, log management, application management, enterprise security and compliance. Splunk, 2005. URL https://www.splunk.com/.

[41] W3techs. Comparison of the usage of apache vs. nginx vs. microsoft-iis for websites. Apache vs. Nginx vs. Microsoft-IIS usage statistics, 2009. URL https://w3techs.com/technologies/comparison/ws-apache,ws-microsoftiis,ws-nginx.

[42] W3techs. Historical yearly trends in the usage of web servers for websites. Historical yearly trends in the usage of web servers, April 2017, 2010. URL https://w3techs.com/technologies/history_overview/web_server/ms/y.

[43] F. Weimer. Passive DNS Replication. In *Conference on Computer Security Incident Handling*, June 2005.

[44] Wikipedia. Transport layer security, 2017. URL https://en.wikipedia.org/wiki/Transport_Layer_Security.