From Patches to Honey-Patches: Lightweight Attacker Misdirection, Deception, and Disinformation*

Frederico Araujo Kevin W. Hamlen
The University of Texas at Dallas
{frederico.araujo, hamlen}@utdallas.edu

Sebastian Biedermann Stefan Katzenbeisser
Technische Universität Darmstadt
{biedermann, katzenbeisser}@seceng.informatik.tu-darmstadt.de

ABSTRACT

Traditional software security patches often have the unfortunate side-effect of quickly alerting attackers that their attempts to exploit patched vulnerabilities have failed. Attackers greatly benefit from this information; it expedites their search for unpatched vulnerabilities, it allows them to reserve their ultimate attack payloads for successful attacks, and it increases attacker confidence in stolen secrets or expected sabotage resulting from attacks.

To overcome this disadvantage, a methodology is proposed for reformulating a broad class of security patches into *honey-patches*—patches that offer equivalent security but that frustrate attackers' ability to determine whether their attacks have succeeded or failed. When an exploit attempt is detected, the honey-patch transparently and efficiently redirects the attacker to an unpatched decoy, where the attack is allowed to succeed. The decoy may host aggressive software monitors that collect important attack information, and deceptive files that disinform attackers. An implementation for three production-level web servers, including Apache HTTP, demonstrates that honey-patching can be realized for large-scale, performance-critical software applications with minimal overheads.

Categories and Subject Descriptors: D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement; D.4.6 [Operating Systems]: Security and Protection; K.6.5 [Management of Computing and Information Systems]: Security and Protection—Unauthorized access

Keywords: Intrusion detection and prevention; Honeypots

1. INTRODUCTION

Patching continues to be perhaps the most ubiquitous and widely accepted means for addressing newly discovered security vulnerabilities in commodity software products. Microsoft alone released over 100 security bulletins spanning 330 separate vulnerabilities in its products during 2013 [33]. However, despite the increasingly prompt availability of security patches, a majority of attacks in the wild continue to exploit vulnerabilities that are known and for which a patch exists [5, 9, 23]. This is in part because patch adoption is not immediate, and may be slowed by various considerations, such as patch compatibility testing, in some sectors.

As a result, even determined, resourceful attackers often probe and exploit unpatched, patchable vulnerabilities in their victims. For example, a 2013 security audit of the U.S. Department of Energy revealed that 60% of DoE desktops lacked critical patch updates, leading to a compromise and exfiltration of private information on over 100,000 individuals [22]. The prevalence of unpatched systems has led to tools and technologies via which attackers can quickly derive unique, previously unseen exploits from patches [12], allowing them to infiltrate vulnerable systems.

The obvious solution is, of course, to obey the golden rule of "patch early, patch often." However, once applied, typical security patches have a significant drawback—they advertise that the system is patched. For example, a request that yields garbage output from an unpatched server, but yields an error message from a patched server, readily divulges whether the server is vulnerable. Attackers can quickly and efficiently probe such servers for known vulnerabilities to discover patching lapses and prepare potent attacks.

To misdirect such attackers, we propose patching vulnerabilities in such a way that failed exploits appear to succeed. This frustrates attackers' ability to discern which apparent vulnerabilities will actually divulge secrets or do damage once fully exploited. We refer to such a patch as a *honey-patch*. Honey-patches offer equivalent security to conventional patches, but respond to attempted exploits by transparently redirecting the attacker's connection to a carefully isolated *decoy* environment. The decoy is unpatched, allowing the attack to succeed, but gathers information about the threat (e.g., collecting and analyzing previously unseen malware), and feeds disinformation to the attacker in the form of falsified data (cf., [11, 48, 62]).

As an illustration of where honey-patching could be useful, consider the digital infrastructure mandated by the U.S. Patient Protection and Affordable Care ("Obamacare") Act [52]. The act entails the deployment of federal- and state-level web servers that sell health care plans. These servers have been identified as inviting targets of directed cyber-attacks, since they receive a wealth of personally identifying information that could be abused for identity theft and fraud [21]. Patching these servers in the conventional way protects against known

*This research was supported by ONR grant N00014-14-1-0030, AFOSR grant FA9550-14-1-0173, NSF grant 1054629, CASED and EC-SPRIDE Darmstadt, and BMBF grant 01C12S01V.

CCS'14, November 3–7, 2014, Scottsdale, Arizona, USA.

http://dx.doi.org/10.1145/2660267.2660329.

Listing 1: Abbreviated patch for Heartbleed

```
+ if (1 + 2 + payload + 16 > s->s3->rrec.length)
+   return 0;  // silently discard
```

exploits, but facilitates an attacker's probing efforts until an unpatched vulnerability is found. Network-level filters redirect known malware to honey-servers, but may not catch custom malware payloads whose exploitive behavior is only detected at execution.

However, honey-patching the servers and stocking the decoy environments with false information that has been red-flagged in identity theft databases greatly increase risk for attackers who manage to bypass other safeguards to reach the decoy. Apparently successful attacks may then yield tainted information that could lead to an arrest when used. Even if the deception is eventually uncovered, defenders gain valuable threat information by aggressively monitoring the most dangerous attacks, and attacker reconnaissance efforts are impeded.

While the concept of honey-patching is straightforward, realizing it in practice is not. To demonstrate and evaluate its feasibility, we present an implementation for three high-performance web servers: Apache HTTP, Lighttpd, and Nginx. Our implementation, RedHerring, Redirects Exploits to Deceptive Honeypot Environments for counteRReconnaissance and INformation Gathering. We chose Apache as our flagship case-study due to its complexity (2.2M SLOC) and its use in many security-sensitive contexts. For example, most Obamacare web sites presently use it.

Our work includes the following contributions:

- We outline a strategy for easily reformulating many vendor-supplied, source-level patches into equally secure honey-patches that raise attacker risk and uncertainty.
- We introduce a light-weight, resource-efficient, and fine-grained approach to transparently fork attacker connections to a sandboxed decoy devoid of secrets.
- Our decoy generation incorporates a novel technique for efficient *in-memory* redaction of secrets that could otherwise be abused by attackers.
- Implementations and evaluations for three production web servers demonstrate that the approach is feasible for large-scale, performance-critical software with minimal overheads for legitimate users.

Section 2 first outlines the honey-patching process and presents our system design. Section 3 describes the architecture in greater detail, and Section 4 explores the central challenge of efficient, live redirection of attacker sessions to decoys. Our implementation is summarized in Section 5 and evaluated in Section 6. Discussion and related work are presented in Sections 7 and 8, respectively, and Section 9 concludes with a summary of outcomes and future directions.

2. SYSTEM OVERVIEW

We first outline the concept of a honey-patch, and then describe primary challenges and corresponding high-level design decisions for honey-patching. Finally, we summarize important technologies that undergird our implementation approach.

2.1 From Patches to Honey-Patches

Listing 1 shows an abbreviated patch in diff style for the Heartbleed OpenSSL buffer over-read vulnerability (CVE-2014-0160) [14]—one of the most significant vulnerability

Listing 2: Honey-patch for Heartbleed

```
1    if (1 + 2 + payload + 16 > s->s3->rrec.length)
2  + {
3  +   hp_fork();
4  -   return 0;  // silently discard
5  +   hp_skip(return 0);  // silently discard
6  + }
```

disclosures in recent history, affecting a majority of then-deployed web servers, including Apache. The patch introduces a conditional that validates SSL/TLS heartbeat packets, declining malformed requests. Prior to being patched, attackers could exploit this bug to acquire sensitive information from many web servers.

This patch exemplifies a common vulnerability mitigation: dangerous inputs or program states are detected via a boolean test, with positive detection eliciting a corrective action. The corrective action is typically readily distinguishable by attackers—in this case, the attacker request is silently declined. As a result, the patched and unpatched programs differ only on attack inputs, making the patched system susceptible to probing. Our goal in this work is to introduce a strategy whereby administrators of products such as Apache can easily transform such patches into honey-patches, whose corrective actions impede attackers and offer strategic benefits to defenders.

Toward this end, Listing 2 presents an alternative, honey-patched implementation of the same patch. In response to a malformed input, the honey-patched application forks itself onto a confined, ephemeral, decoy environment, and behaves henceforth as an unpatched, vulnerable version of the software. Specifically, line 3 forks the user session to a decoy container, and macro `hp_skip` in line 5 elides the rejection in the decoy container so that the attack appears to have succeeded. Meanwhile, the attacker session in the original container is safely terminated (having been forked to the decoy), and legitimate, concurrent connections continue unaffected.

Observe that the differences between the patch and the honey-patch are quite minor, except for the fixed cloning infrastructure that the honey-patch code references, and that can be maintained separately from the server code. This allowed us to formulate a Heartbleed honey-patch within hours of receiving the vulnerability disclosure on April 7, facilitating a quick, aggressive response to the threat [53]. In general, only a superficial understanding of many patches is required to convert them to honey-patches of this form. (A more systematic study of honey-patchable patches is presented in §6.) However, the cloning infrastructure required to facilitate efficient, transparent, and safe redirection to decoys demands a careful design.

2.2 Challenges & Design Decisions

Although the honey-patching approach described above is simple on the surface, there are many significant security and performance challenges that must be surmounted to realize it in practice. For example, a naïve forking implementation copies any secrets in the victim process's address space, such as encryption keys of concurrent sessions, over to the child decoy. In a honey-patching framework this would be disastrous, since the attack is allowed to succeed in the decoy, thereby giving the attacker potential access to any secrets it may contain.

Moreover, practical adoption requires that honey-patches (1) introduce almost no overhead for legitimate users, (2) perform well enough for attackers that attack failures are not placarded, and (3) offer high compatibility with software that boasts aggressive multi-processing, multi-threading, and active connection migration across IPs. Solutions must be sufficiently modular and generic that administrators require only a superficial, high-level understanding of each patch's structure and semantics to reformulate it as an effective honey-patch. Specifically, we envision the following practical requirements:

1. Remote forking of attacker sessions must happen live, with no perceptible disruption in the target application; established connections must not be broken.
2. Decoy deployment must be fast, to avoid offering overt, reliable timing channels that advertise the honey-patch.
3. All sensitive data must be redacted before the decoy resumes execution.

Together, these requirements motivate three main design decisions. First, the required time performance precludes system-level cloning (e.g., VM cloning [13]) for session forking; instead, we employ a lighter-weight, finer-grained alternative based on process migration through *checkpoint-restart* [41]. To scale to many concurrent attacks, we use an *OS-level virtualization* technique to deploy forked processes to decoy containers, which can be created, deployed, and destroyed orders of magnitude faster than other virtualization techniques, such as full virtualization or para-virtualization [60].

Second, our approach to remote session forking benefits from the synergy between mainstream Linux kernel APIs and user-space tools, allowing for a small freezing time of the target application. To maintain established connections when forking, we have conceived and implemented a *connection relocation* procedure that allows for transparent session migration.

Third, to guarantee that successful exploits do not afford attackers access to sensitive data stored in application memory, we have implemented a *memory redaction* and light-weight synchronization mechanism during forking. This censors sensitive data from process memory before the forked (unpatched) session resumes. Forked decoys host a deceptive file system that omits all secrets, and that can be laced with disinformation to further deceive, delay, and misdirect attackers.

2.3 Threat Model

Attackers in our model submit malicious inputs (HTTP requests) intended to probe and exploit known vulnerabilities on victim web servers. Our system does not defend against exploits of previously unknown (i.e., zero-day) vulnerabilities; such protection is outside our scope.

Although the exploited vulnerabilities are known, we assume that the attack payloads might be completely unique and therefore unknown to defenders. Such payloads might elude network-level monitors, and are therefore best detected at the software level at the point of exploit. We also assume that attackers might use one payload for reconnaissance but reserve another for the final attack. Misleading the attacker into launching the final attack is therefore useful for discovering the final attack payload, which can divulge attacker strategies and goals not discernible from the reconnaissance payload alone.

Attacker requests are processed by a server possessing strictly user-level privileges, and must therefore leverage web server bugs and kernel-supplied services to perform malicious actions, such as corrupting the file system or accessing other users' memory to access confidential data. The defender's ability to thwart these and future attacks stems from his ability to deflect attackers to fully isolated decoys and perform counterreconnaissance (e.g., attack attribution and information gathering).

2.4 Background

Apache HTTP has been the most popular web server since April 1996 [4]. Its market share includes 54.5% of all active websites (the second, Nginx, has 11.97%) and 55.45% of the top-million websites (against Nginx with 15.91%) [42]. It is a robust, commercial-grade, feature-rich open-source software product comprised of 2.27M SLOC mostly in C [44], and has been tested on millions of web servers around the world. These characteristics make it a highly challenging, interesting, and practical flagship case study to test our approach.

Process Checkpoint-restart. Process migration through checkpoint-restart is the act of transferring a running process between two nodes by dumping its state on the source and resuming its execution on the destination. (A mechanism to transfer the state between nodes is assumed.) Recently, there has been growing interest in this problem, especially for high-performance computing [27, 57]. As a result, several emerging tools have been developed to support performance-critical process checkpoint-restart (e.g., BLCR [20], DMTCP [3], and CRIU [18]). Process checkpoint-restart plays a pivotal role in making the honey-patch concept viable. In this work, we have extended CRIU (Checkpoint/Restore In Userspace) [18] with *memory redaction* and transparent *relocation* of TCP connections to restore active attacker sessions in decoys without disrupting the source web server.

Linux Containers. OS-level virtualization allows multiple guest nodes (*containers*) to share the kernel of their controlling host. Linux containers (LXC) [39] implement OS-level virtualization, with resource management via process control groups and full resource isolation via Linux namespaces. This ensures that each container's processes, file system, network, and users remain mutually isolated.

For our purposes, LXC offers a lightweight sandbox that we leverage for attacker session isolation. For efficient container management, we use the overlay file system to deploy containers backed by a regular directory (the *template*) to clone new *overlayfs* containers (decoys), mounting the template's root file system as a read-only lower mount and a new private delta directory as a read-write upper mount. The template used to clone decoys is a copy of the target container in which all sensitive files are replaced with disinformation.

3. ARCHITECTURE

The architecture of RedHerring is shown in Fig. 1. Central to the system is a *reverse proxy* that acts as a transparent proxy between users and internal servers deployed as LXC containers. The *target* container hosts the honey-patched web server instance, and the n decoys form the pool of ephemeral containers managed by the *LXC Controller*. The decoys serve as temporary environments for attacker sessions. Each container runs a *CR-Service* (Checkpoint/Restore) daemon, which exposes an interface controlled by the *CR-Controller* for remote checkpoint and restore.

Honey-patch. The honey-patch mechanism is encapsulated in a tiny C library, allowing for low-coupling between target application and honey-patching logic. The library exposes three API functions:

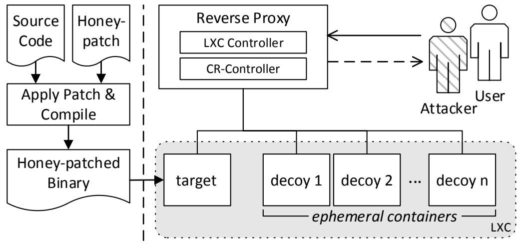


Figure 1: RedHerring system architecture overview

- `hp_init`($pgid, pid, tid, sk$): initialize honey-patch with the process group $pgid$, process pid, thread tid, and socket descriptor sk of the session.
- `hp_fork()`: initiate the attacker session remote forking process, implementing the honey-patching core logic.
- `hp_skip`(c): skip over block c if in a decoy.

Function `hp_init` initializes the honey-patch with the necessary information to handle subsequent session termination and resurrection. It is invoked once per HTTP connection, at the start of the session life cycle. In Apache, this immediately follows acceptance of an HTTP request and handing the newly created session off to a child process or worker thread; in Lighttpd and Nginx, it follows the accept event for new connections.

Listing 3 details the basic steps of `hp_fork`. Line 3 determines the application context, which can be either `target` (denoting the target container) or `decoy`. In a decoy, the function does nothing, allowing multiple attacks within a single attacker session to continue within the same decoy. In the target, a fork is initiated, consisting of four steps: (1) Line 5 registers the signal handler for session termination and resurrection. (2) Line 6 sends a *fork request* containing the attacker session's $pgid$, pid, and tid to the proxy's CR-Controller. (3) Line 7 synchronizes checkpoint and restore of the attacker session in the target and decoy, respectively, and guarantees that sensitive data is redacted from memory before the clone is allowed to resume. (4) Once forking is complete and the attacker session has been resurrected, the honey-patch context is saved and the attacker session resumes in the decoy.

The fork request (step 2) achieves high efficiency by first issuing a system `fork` to create a shallow, local clone of the web server process. This allows event-driven web servers to continue while attacker sessions are forked onto decoys, without interrupting the main event-loop. It also lifts the burden of synchronizing concurrent checkpoint operations, since CRIU injects a Binary, Large OBject (BLOB) into the target process memory space to extract state data during checkpoint (see §4).

The context-sensitivity of this framework allows the honey-patch code to exhibit context-specific behavior: In decoy contexts, `hp_skip` elides the execution of the code block passed as an argument to the macro, elegantly simulating the unpatched application code. In a target context, it is usually never reached due to the fork. However, if forking silently fails (e.g., due to resource exhaustion), it abandons the deception and conservatively executes the original patch's corrective action for safety.

LXC Pool. The decoys into which attacker sessions are forked are managed as a pool of Linux containers controlled by the LXC Controller. The controller exposes two operations to the proxy: *acquire* (to acquire a container from the pool), and *release* (to release back a container to the pool).

Listing 3: `hp_fork` function

```
1 void hp_fork()
2 {
3   read_context();       // read context (target/decoy)
4   if (decoy) return;    // if in decoy, do nothing
5   register_handler();   // register signal handler
6   request_fork();       // fork session to decoy
7   wait();               // wait until fork process has finished
8   save_context();       // save context and resume
9 }
```

Each container follows the life cycle depicted in Fig. 2. Upon receiving a fork request, the proxy acquires the first available container from the pool. The acquired container holds an attacker session until (1) the session is deliberately closed by the attacker, (2) the connection's *keep-alive* timeout expires, (3) the ephemeral container crashes, or (4) a session timeout is reached. The last two conditions are common outcomes of successful exploits. In any of these cases, the container is released back to the pool and undergoes a recycling process before becoming available again.

Recycling a container encompasses three sequential operations: *destroy*, *clone* (which creates a new container from a template in which legitimate files are replaced with honeyfiles), and *start*. These steps happen swiftly for two main reasons. First, the lightweight virtualization implemented by LXC allows containers to be destroyed and started similarly to how OS processes are terminated and created. Second, we deploy our ephemeral containers as overlayfs-based clones, making the cloning step almost instantaneous.

CR-Service. The Reverse Proxy uses the CR-Controller module to communicate with CR-Service daemons running in the background of each container. Each CR-Service implements a *façade* that exposes CR operations to the proxy's CR-Controller through a simple RPC protocol based on Protocol Buffers [28]. To enable fast, OS-local RPC communication, we use IPC sockets (a.k.a., Unix domain sockets).

The CR-Service uses an extended version of CRIU to checkpoint attacker sessions on the target and restore them on decoys. It is also responsible for sanitizing process image files (dump files), which are written to disk during checkpoint, as well as managing attacker session signaling.

Reverse Proxy. The proxy plays a dual role in the honey-patching system, acting as (1) a *transport layer transparent proxy*, and (2) an *orchestrator* for attacker session forking.

As a transparent proxy, its main purpose is to hide the backend web servers and route client requests. To serve each client's request, the proxy server accepts a downstream socket connection from the client and binds an upstream socket connection to the backend HTTP server, allowing HTTP sessions to be processed transparently between the client and the backend server. To keep its size small, the proxy neither manipulates message payloads, nor implements any rules for

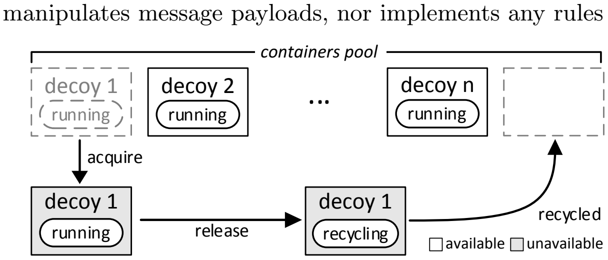


Figure 2: Linux containers pool and decoys life cycle

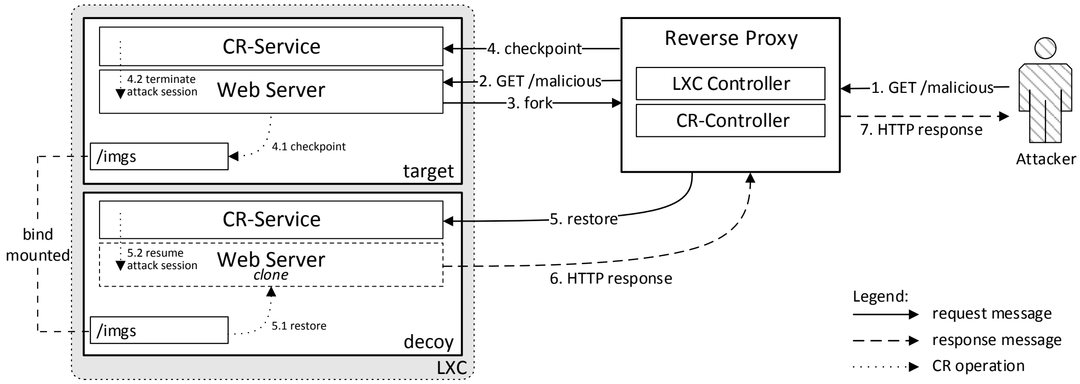


Figure 3: Attacker session forking. Numbers indicate the sequential steps taken to fork an attacker session.

detecting attacks. There is also no session caching. This makes it extremely innocuous and lightweight. We implemented the proxy as a transport-layer reverse proxy to reduce routing overhead and support the variety of protocols operating above TCP, including SSL/TLS.

As an orchestrator, the proxy listens for fork requests and coordinates the attacker session forking as shown in Fig. 3. Under legitimate load, the proxy simply routes user requests to the target and routes server responses to users. However, attack inputs elicit the following alternate workflow:

Step 1: The attacker probes the server with a crafted request (denoted by `GET /malicious` in Fig. 3).

Step 2: The reverse proxy transparently routes the request to the backend target web server.

Step 3: The request triggers the honey-patch (i.e., when the honey-patch detects an attempted exploit of the patched vulnerability) and issues a fork request to the reverse proxy.

Step 4: The proxy's CR-Controller processes the request, acquires a decoy from the LXC Pool, and issues a checkpoint RPC request to the target's CR-Service. The CR-Service
- **4.1:** checkpoints the running web server instance to the `/imgs` directory; and
- **4.2:** signals the attacker session with a termination code, gracefully terminating it.

Step 5: Upon checkpoint completion, the CR-Controller commands the decoy's CR-Service to restore the dumped web server images on the decoy. The CR-Service then
- **5.1:** restores a clone of the web server from the dump images located in the `/imgs` directory; and
- **5.2:** signals the attacker session with a resume code, and cleans the dump data from `/imgs`.

Step 6: The attacker session resumes on the decoy, and a response is sent back to the reverse proxy.

Step 7: The reverse proxy routes the response to the attacker.

Throughout this workflow, the attacker's session forking is completely transparent to the attacker. To avoid any substantial overhead for transferring files between target and decoys, we adopt the strategy of bind-mounting each decoy's `/imgs` folder to the target's `/imgs` directory. After the session has been forked to the decoy, it behaves like an unpatched server, making it appear that no redirection has taken place and the original probed server is vulnerable.

4. SESSION REMOTE FORKING

At the core of our architecture is the capability of remote forking an attacker session to a decoy through checkpoint and restore of the target server. To this end, we have extended CRIU [18] with a memory redaction procedure performed during checkpoint to protect sensitive data of legitimate users, and a transparent connection relocation mechanism to restore TCP connections in the destination decoy without stopping the target server. We name this extended version CRIU$_\mathrm{m}$.

4.1 Checkpoint

The checkpoint procedure takes place in the target container and is initiated when the CR-Service receives a checkpoint request. The request includes the process group leader `$pgid`, attacker process `$pid`, and attacker thread `$tid`.

The CR-Service passes this information to our CRIU$_\mathrm{m}$ checkpoint interface, which in turn: (1) uses the `/proc` file system to collect file descriptors (`/proc/$pgid/fd` and `/proc/$pgid/fdinfo`), pipe parameters, and memory maps (`/proc/$pgid/maps`) for the process group; (2) walks through `/proc/$pgid/task/` and gathers child processes recursively to build the process tree; (3) locks the network by adding netfilter rules and collecting socket information; (4) uses `ptrace` (with `PTRACE_SEIZE`) to attach to each child (without stopping it) and collect VMA areas, the task's file descriptor numbers, and core parameters such as registers; (5) injects a BLOB code into the child address space to collect state information such as memory pages; (6) performs memory redaction using `$pid` and `$tid`; (7) uses `ptrace` to remove the injected code from the child process and continues until all children have been traced; (8) unlocks network using netfilter, and finishes the procedure by writing the process tree image files to `/imgs/$tid/`.

At this point, CRIU$_\mathrm{m}$ returns to the caller, the web server is running, and the attacker thread waits to be signaled. The CR-Service then sends a termination signal to the attacker thread, which terminates itself gracefully in the target web server. This successfully completes the checkpoint request, and the CR-Service sends a `success` status response to the CR-Controller.

We next examine the memory redaction step in greater detail, to explain how sensitive, in-memory data is safely replaced with decoy data during the fork.

Memory Redaction. Were session cloning performed in the typical, rote fashion of copying all bytes, attackers who successfully hijack decoys could potentially view any confidential data copied from the memory space of the original process (e.g., in a multi-threaded setting). Sophisticated attacks could thus glean sensitive information about other users previously or concurrently connected to the original server process, if such information is cloned with the process. In web servers, such sensitive information includes IP addresses of other users, request histories, and information about encrypted connections. It is therefore important to redact these secrets during cloning.

We therefore introduce a memory redaction procedure that replaces sensitive data with specially forged, anonymous data during cloning. Since every server application has different forms of sensitive data stored in slightly different ways, our solution is a general-purpose tool that must be specialized to each server product by an administrator prior to deployment. In the case of Apache, we focus on redaction of user request data, session data, and SSL context data, which Apache records in a few data structures stored in memory for each user session. For instance, Apache's `request_rec` struct stores request histories. Other servers store such data in similar ways, but we omit their discussion due to limited space.

A brute force strategy for memory redaction is to search the entire process memory space to match and replace sensitive data. Such a strategy does not perform well. Instead, we leverage the fact that most security-relevant data are stored in struct variables in heap or stack memory, allowing us to narrow the search space significantly. Freed memory is included in the search. For efficiency, our redactor replaces these structures with anonymous data having exactly the same length and characteristics. For example, IP addresses in `request_rec` are replaced with strings having the same length that are also valid IP addresses, but randomly generated. This yields a realistic, consistent process image that can continue running without errors (save possibly for effects of the attack).

The redaction is implemented as a step of the checkpoint procedure, so that the image files temporarily created during process checkpoint and shared with decoys do not contain any sensitive information that could be potentially abused by attackers. Secrets are redacted from all session-specific structures except the attacker's, allowing the attacker's session to continue uninterrupted.

We initially implemented memory redaction as a separate operation applied to the image files generated by CRIU. While this seemed attractive for avoiding modification of CRIU, it exhibited poor performance due to reading and writing the image files multiple times. Our revised implementation therefore realizes redaction as a streaming operation within CRIU's checkpointing algorithm. In-lining it within checkpointing avoids reloading the process tree images into memory for redaction. In addition, redacting secrets before dumping the process images avoids ever placing secrets on disk.

4.2 Restore

Upon successful completion of a checkpoint operation, the CR-Controller sends a request to the decoy's CR-Service into which the attacker session is to be forked. In addition to `$pgid`, `$pid`, and `$tid`, the body of the restore request contains a callback `port` that has been dynamically assigned by the reverse proxy to hold the new back-end connection associated with the attacker session. Once the request is parsed, the CR-Service passes this information as parameters to the CRIU_m restore interface, which (1) reads the corresponding process tree from `/imgs/$tid/`; (2) uses the `clone` system call to start each dumped process found in the process tree with its original process ID; (3) restores file descriptors and pipes to their original states, and executes relocation of `ESTABLISHED` socket connections; (4) injects a BLOB code into the process address space to recreate the memory map from the dumped data;

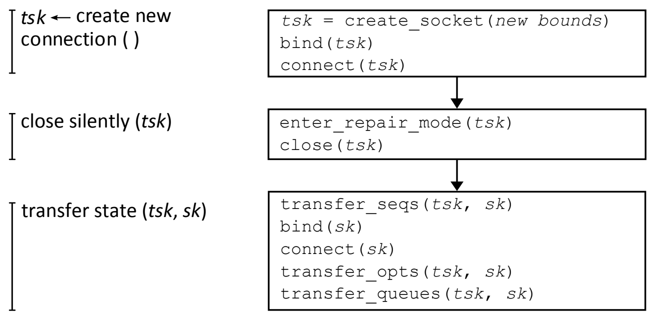


Figure 4: Procedure for TCP connection relocation

(5) removes the injected binary, and resumes the execution of the application via the `rt_sigreturn` system call.

At this point, CRIU_m returns to the caller, the forked instance is running on the decoy, and the attacker thread waits to be signaled. The CR-Service sends a resume signal to the attacker thread, which allows it to resume request processing. This completes the restore request, and the CR-Service sends a `success` response to the CR-Controller. Subsequent attacker requests are relayed to the decoy instead of the target, as discussed in §3. Next, we discuss details of the TCP connection relocation procedure.

Established Connection Relocation. Target and decoys are fully isolated containers running on separate namespaces. As a result, each container is assigned a unique IP in the internal network, which affects how we move active connections from the target to a decoy. Since CRIU was not implemented with this use case in mind, we extended it to support relocation of TCP connections during process restoration. In what follows, we explain how we approached this problem.

The reverse proxy always routes legitimate user connections to the target; hence, there is no need to restore the state of connections for these users when restoring the web server on a decoy. We simply restore legitimate connections to "drainer" sockets, since we have no interest in maintaining legitimate user interaction with the decoys. This ensures that the associated user sessions are restored to completion without interrupting the overall application restoration.

Conversely, the attacker connection must be restored to its dumped state when switching the attacker session to a decoy. This is important to avoid connection disruption and to allow transparent session migration (from the perspective of the attacker). To accomplish this, our proxy dynamically establishes a new back-end TCP connection between proxy and decoy containers in order to hold the attacker session communication. Moreover, a mechanism based on *TCP repair options* [16] is employed to transfer the state of the original attacker's session socket (bound to the target IP address) into the newly created socket (bound to the decoy IP address).

Figure 4 describes the connection relocation mechanism, implemented as a step of the attacker's session restore process. At process checkpoint, the state information of the original socket `sk` is dumped together with the process image (not shown in the figure). This includes connection bounds, previously negotiated socket options, sequence numbers, receiving and sending queues, and connection state. During process restore, we relocate the connection to the assigned decoy by (1) connecting a new socket `tsk` to the proxy `$port` given in the restore request, (2) setting `tsk` to *repair mode* and silently closing the socket (i.e., no `FIN` or `RST` packages are sent to the

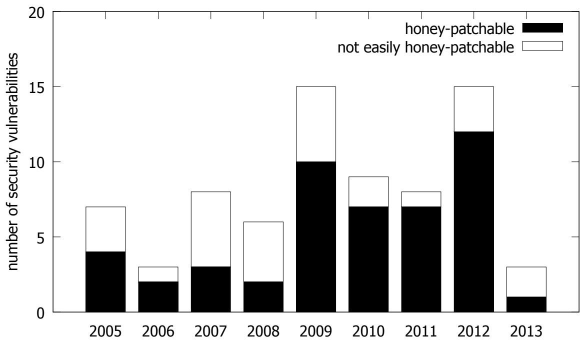


Figure 5: Apache honey-patchable vulnerabilities

remote end), and (3) transferring the connection state from `sk` to `tsk` in repair mode. Once the new socket `tsk` is handed over to the restored attacker session, the relocation process has completed and communication resumes, often with an HTTP response being sent back to the attacker.

5. IMPLEMENTATION

We have developed an implementation of RedHerring for the 64-bit version of Linux (kernel 3.11 or above). The implementation consists of five components: the honey-patch library, the LXC-Controller, the CR-Controller, the CR-Service, and the reverse proxy. The honey-patch library provides the tiny API required for triggering the honey-patching mechanism. Its implementation consists of about 270 lines of C code that uses no external libraries or utilities. The reverse proxy routes HTTP/S requests in accordance with the behavior described in §3. Its implementation is fully asynchronous and consists of about 325 lines of node.js JavaScript code. The CR-Controller is implemented as an external C++ module to the proxy, and consists of about 450 lines of code that uses Protocol Buffers to communicate with the CR-Service. Similarly, the LXC-Controller is implemented as an external node.js library consisting of about 190 lines of code. The CR-Service receives CR requests from the CR-Controller and uses $\mathrm{CRIU_m}$ to coordinate process checkpoint and restore. Its implementation comprises about 525 lines of C code. Our extensions to $\mathrm{CRIU_m}$ add about 710 lines of C code to the original CRIU tool.

6. EVALUATION

This section discusses the applicability of honey-patching and investigates performance characteristics of the session live migration scheme implemented by RedHerring. First, we survey the past nine years of Apache's security reports to assess the proportion of security patches that are amenable to our honey-patching technique. Then, we investigate the effect of session migration on malicious attack HTTP response times and report measurements of the impact of concurrent attacks on legitimate HTTP request round-trip times. Finally, we compare the performances of the honey-patched versions of Apache, Lighttpd, and Nginx.

All experiments were performed on a quad-core virtual machine (VM) with 8 GB RAM running 64-bit Ubuntu 14.04 (Trusty Tahr). Each LXC container running inside the VM was created using the official LXC Ubuntu template. We limited resource utilization on decoys so that a successful attack does not starve the host VM. The host machine is an Intel Xeon E5645 desktop running 64-bit Windows 7.

Listing 4: Abbreviated patch for CVE-2013-1862

```
logline = apr_psprintf(r->pool, ...,
  ...
- ap_get_server_name(r),
+ ap_escape_logitem(r->pool, ...(r)),
  ...
```

6.1 Honey-patchable Patches

Our strategy sketched in §2.1 for transforming patches into honey-patches is more easily applied to some patches than others. In general, patches that have a clear, boolean decision point where patched and unpatched application behavior diverge are best suited to our approach, whereas patches that introduce deeper changes to the application's control-flow structure or data structures may require correspondingly deeper knowledge of the patch's semantics to reformulate as a honey-patch.

To assess the practicality of honey-patching, we surveyed all vulnerabilities officially reported by the Apache HTTP web server project between 2005 and 2013. We systematically examined each security patch file and corresponding source code to determine its amenability to honey-patching. Figure 5 reports the results. Overall, we found that 49 out of 75 patches analyzed (roughly 65%) are easily transformable into honey-patches. This corroborates the intuition that most security vulnerabilities are patched with some small check, usually one that performs input validation [12].

Listing 4 shows an example of a patch (simplified for brevity) for which honey-patching is not elementary. The patch replaces the insecure method `ap_get_server_name` with an alternate one (`ap_escape_logitem`) that performs input sanitization. The sanitization step lacks any boolean decision point where exploits are detected; it instead performs a string transformation that replaces dangerous inputs with non-dangerous ones. Thus, it is not obvious where to position the forking operation needed for a honey-patch.

However, even in the case of Listing 4, we note that honey-patching is still possible, given a sufficiently comprehensive understanding of the patch's semantics. In particular, this patch could be converted to a honey-patch by retaining both the sanitizing and non-sanitizing implementations and comparing the resulting strings. If the strings differ, the honey-patch forks the session to a decoy. Note that not every input sanitization patch can be honey-patched in this way, since some sanitization procedures modify even non-dangerous inputs. Thus, patches of this sort were conservatively classified as not easily honey-patchable in our study, since they require greater effort to honey-patch.

Experimental validation. To evaluate honey-patching's effectiveness in diverting attackers to decoys, we tested RedHerring with different honey-patched Apache releases. Table 1 summarizes the tested versions of Apache and corresponding vulnerabilities that we successfully exploited. For each vulnerability, we tested the system on non-malicious inputs and verified that RedHerring does not fork any attacker sessions. Then we exploited honey-patched vulnerabilities, and verified that the system behaves as a vulnerable decoy server in response to the attack inputs.

Apache 2.2.21 allows the inadvertent exposure of internal resources to remote users who send specially crafted requests (CVE-2011-3368). For example, the request `GET @private.com/topsecret.pdf HTTP 1.1` may result in an exposure of unpatched servers. The security patch for this vulnerability modifies `protocol.c` to send an HTTP 400

Table 1: Honey-patched security vulnerabilities for different versions of the Apache Web Server

Version	CVE-ID	Description
2.2.21	CVE-2011-3368	Improper URL validation
2.2.9	CVE-2010-2791	Improper timeouts of keep-alive connections
2.2.15	CVE-2010-1452	Bad request handling
2.2.11	CVE-2009-1890	Request content length out of bounds
2.0.55	CVE-2005-3357	Bad SSL protocol check

response if the request URI is not an absolute path. Our honey-patch forks to a decoy instead.

Similarly, we honey-patched and tested CVE-2010-1452 and CVE-2009-1890, which involve improper HTTP request sanitization. CVE-2010-1452 exposes a request handling problem in which requests missing the `path` field may cause the worker process to segfault, inviting potential DOS attacks. CVE-2009-1890 exposes another type of DOS vulnerability in which a sufficiently long HTTP request may lead to memory exhaustion.

CVE-2010-2791 allows us to test RedHerring against attacks exploiting vulnerabilities related to keep-alive connections. In this particular case, a bug neglects closing the back-end connection if a timeout occurs when reading a response from a persistent connection, which allows remote attackers to obtain a potentially sensitive response intended for a different client. Finally, CVE-2005-3357 exposes a bad SSL protocol check that allows an attacker to cause a DOS if a non-SSL request is directed to an SSL port.

6.2 Performance Benchmarks

This section evaluates RedHerring's performance. Our objectives are two-fold: to determine the performance overhead imposed upon sessions forked to decoys (i.e., the impact on malicious users), and to estimate the impact of honey-patching on the overall system performance (i.e., its impact on legitimate users). To obtain baseline measurements that are independent of networking overhead, the experiments in this section are executed locally on a single-node virtual machine using default Apache settings. Performance is measured in terms of HTTP request round-trip time.

Session forking overhead. As expected, forking attacker sessions from target to decoy containers is the main source of performance overhead in RedHerring. To estimate its impact on attacker response times, we crafted and sent malicious requests to the server in order to trigger its internal honey-patching mechanism, and measured the request round-trip time of each individual request. For accuracy, we waited for the completion of each request before sending another one.

Since Apache allocates requests (including their content) on the heap, payload size directly impacts the amount of data to be dumped in each checkpoint operation. It is therefore important to experiment using varying sizes of malicious HTTP request payload data (from 0 KB to 36 KB, in steps of 1.2 KB). Also, to estimate the response time overhead incurred from the memory redaction process, we executed our tests twice, with memory redaction enabled and disabled.

Figure 6a shows the encouraging results of this experiment. Malicious HTTP request round-trip times tend to remain almost constant as payload size increases. This desirable relationship can be explained by two reasons. First, CRIU's approach of copying process memory pages into dump files during checkpoints is extremely efficient. It involves a direct copy of data between file descriptors in kernel space using the `splice` system call. As a consequence, the target memory pages are never buffered into user space. Second, our approach to memory redaction leverages the fact that Apache stores session data in well-defined structs (avoiding having multiple copies in memory) to locate and redact it directly into the dump images while they are being generated by the checkpoint process. Our initial efforts to implement redaction after checkpointing exhibited far poorer performance, leading to this more efficient solution.

Overall, the round-trip times of malicious HTTP requests incur a constant overhead of approximately 0.25 seconds due to memory redaction. (When memory redaction is used, the average request takes approximately 0.40 seconds; but when disabled, it takes 0.15 seconds.) While possibly significant (depending on networking latencies), we emphasize that this constant overhead *only impacts malicious users*, as demonstrated by the next experiment.

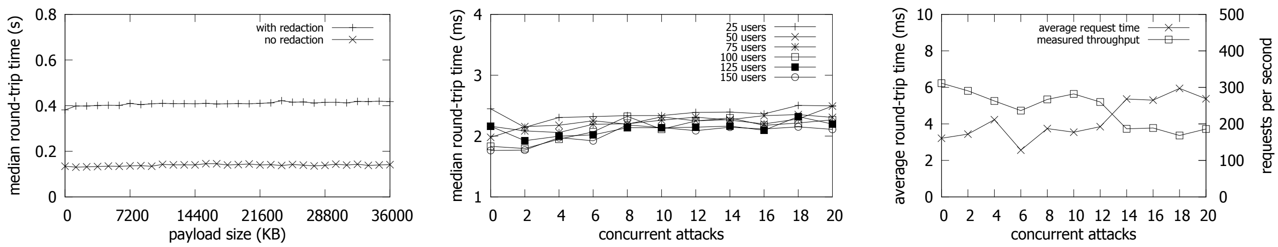
Overall system overhead. To complete our evaluation, we tested RedHerring on a wide variety of workload profiles consisting of both legitimate users and attacker sessions on a single node. In this experiment, we wrote a small Python script modeling every user and attacker as a separate worker thread triggering legitimate and malicious HTTP requests, respectively. We chose the request payload size to be 2.4 KB, based on the median of KB per request measured by Google web metrics [29]. To simulate different usage profiles, we tested our system with 25–150 concurrent users, with 0–20 attackers.

Figure 6b plots our results. Observe that for the various profiles analyzed, the HTTP request round-trip times remain approximately constant (ranging between 1.7 and 2.5 milliseconds) when increasing the number of concurrent malicious requests. This confirms that adding honey-patching capabilities has negligible performance impact on legitimate requests and users relative to traditional patches, even during concurrent attacks. It also confirms our previous claims regarding the small freezing window necessary to checkpoint the target application.

Finally, this also shows that RedHerring can cope with large workloads. In this experiment, we have assessed its baseline performance considering only one instance of the target server running on a single node virtual machine. In a real setting, we can deploy several similar instances using a web farm scheme to scale up to thousands of users, as we show next.

Stress testing. To estimate the throughput of our system and test its scalability properties, we developed a small HTTP load balancer in node.js to round-robin requests between three-node VMs, each hosting one instance of Apache deployed on RedHerring. In this experiment, we used *ab* (Apache HTTP server benchmarking tool) to create a massive workload of legitimate users (more than 5,000 requests in 10 threads) for different attack profiles (0 to 20 concurrent attacks). Each VM is configured with a 2 GB RAM and one quad-core processor. The load balancer and the benchmark tool run on a separate VM on the same host machine. Apache runs with default settings (i.e., no fine tuning has been performed).

As Figure 6c illustrates, the system can handle the strenuous workload imposed by our test suite. The average request time for legitimate users ranged from 2.5 to 5.9 milliseconds, with



(a) $3 \leq \sigma \leq 15\ ms,\ n = 10$ (b) $0.6 \leq \sigma \leq 35,\ n = 10$ (c) $0.3 \leq \sigma \leq 1.2,\ 25 \leq \sigma \leq 94$

Figure 6: Performance benchmarks. (a) Effect of payload size on malicious HTTP request round-trip time. (b) Effect of concurrent attacks on legitimate HTTP request round-trip time on a single-node VM. (c) Stress test illustrating request throughput for a 3-node, load-balanced RedHerring setup (workload $\approx$ 5K requests).

measured throughput ranging from 169 to 312 requests per second. In typical production settings we would expect this delay to be amortized by the network latency (usually on the order of several tens of milliseconds). This result is important because it demonstrates that honey-patching can be realized for large-scale, performance critical software applications with minimal overheads for legitimate users.

6.3 Web Servers Comparison

We also tested RedHerring on Lighttpd [38] and Nginx [43], web servers whose designs are significantly different from Apache. The most notable difference lies in the processing model of these servers, which employs non-blocking systems calls (e.g., select, poll, epoll) to perform asynchronous I/O operations for concurrent processing of multiple HTTP requests. In contrast, Apache dispatches each request to a child process or thread [45]. Our success with these three types of server evidences the versatility of our approach.

Figure 7 shows our results. In comparison to Apache, session forking performed considerably better on Lighttpd and Nginx (ranging between 0.092 seconds without memory redaction and 0.156 seconds with redaction). This is mainly because these servers have smaller process images, reducing the amount of state to be collected and redacted during checkpointing.

7. DISCUSSION

Selective honey-patching. Our work evaluates the feasibility of honey-patching as realistic application, but raises interesting questions about how to evaluate the strategic advantages or disadvantages of honey-patching various specific vulnerabilities. For example, some patches close vulnerabilities by adding new, legitimate software functionalities. Converting such patches to honey-patches might be inadvisable, since it might treat uses of those new functionalities as attacks. In general, honey-patching should be applied judiciously based on an assessment of attacker and defender risk. Future work should consider how to reliably conduct such assessments. Similarly, honey-patching can be applied selectively to simulate different software versions and achieve versioning consistency.

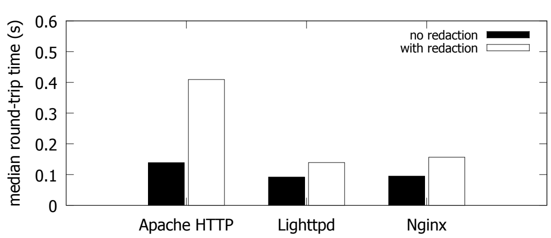


Figure 7: Malicious HTTP request round-trip times for different web servers ($6 \leq \sigma \leq 11\ ms,\ n = 20$)

Automation. Our implementation approach offers a semi-manual process for transforming patches into honey-patches. An obvious next step is to automate this by incorporating it into a rewriting tool or compiler. One interesting challenge concerns the question of how to audit or validate the secret redaction step for arbitrary software. Future research should consider facilitating this by applying language-based information flow analyses (cf., [47]).

Active Defense. Honey-patching enhances the current realm of weaponized software by placing defenders in a favorable position to deploy offensive techniques for reacting to attacks. For example, decoys provide the ideal environment for implementing stealthy traps to disinform attackers and report precisely what attacks are doing in real-time [17], and further insight into the attackers' *modus operandi* can be gained by forging and acting upon decoy data. There is existing work in this direction in DARPA's Mission-oriented Resilient Clouds (MRC) program [55].

Deception. The effectiveness of a honey-patch is contingent upon the deceptiveness of decoy environments. Prior work has investigated the problem of how to generate and maintain convincing honey-data for effective attacker deception (e.g., [11, 48, 54, 62]), but there are other potential avenues of deception discovery that must be considered.

Response times are one obvious channel of possible discovery that must be considered. Cloning is efficient but still introduces non-zero response delay for attackers. By collecting enough timing statistics, attackers might try to detect response delays to discern honey-patches. Our ongoing work is focusing on improving the efficiency of the memory redaction step, which is the source of most of the this delay.

In addition, RedHerring's deceptiveness against discovery through response delays is aided by the plethora of noisy latency sources that most web servers naturally experience, which tend to eclipse the relatively small delays introduced by honey-patching [49]. Unpatched, vulnerable servers often respond slower to malicious inputs than to normal traffic [25, 58], just like honey-patched servers. This suggests that detecting honey-patches by probing for delayed responses to attacks may yield many false positives for attackers. If criminals react to the rise of honey-patching by cautiously avoiding attacks against servers that respond slightly slower when attacked, many otherwise successful attacks will have been thwarted.

Alternatively, attackers who take full control of decoys can potentially read and reverse-engineer the process image's binary code to discover the honey-patch (e.g., by injecting malicious code that reads the process binary and finds the call to `hp_fork`). While possible with enough effort, we believe this is nevertheless a significant burden relative to the much easier task of detecting failed exploits against traditionally patched systems. The emergence of artificial software diversity (e.g., [30]) and fine-grained binary randomization tools (e.g., [59]) has made it increasingly difficult to quickly and reliably reverse-engineer arbitrary binary process images. Future work should consider raising the bar further by unloading prominent libraries, such as the honey-patch library, during cloning.

Additionally, RedHerring's decoy environments are constructed to look identical to real distributed web servers from inside the container; for example, many real web servers use LXC containers that look like the decoy LXC containers. Therefore, distinguishing decoys from real web servers on the basis of environmental details (e.g., through `init` process control groups) is difficult for attackers. Although resource exhaustion attacks (e.g., flooding) can cause RedHerring to run out of decoy resources (e.g., containers), this outcome is difficult for attackers to distinguish from running the same attacks against a non-honey-patched server; both result in observationally similar resource exhaustions.

Likewise, real-time behavior of the decoy inevitably differs from the target due to the lack of other, concurrent connections. With a long enough observation period, attackers can reliably detect this difference (cf., [24]). We mitigate this by constraining decoy lifetimes with a timeout. This resembles unpatched servers that automatically reset when a crash or freeze is detected, and therefore limits the attacker's observations of real-time connection activity without revealing the honey-patch or limiting the attacker's access to decoy data or honeyfiles.

Detection granularity. One foundational assumption of our work is that some attacks are not identifiable at the network or system level before they do damage. Thus, detection approaches that monitor network or system logs for malicious activity are not a panacea. For example, encrypted, obfuscated payloads buried in a sea of encrypted connection data, or those that undertake previously unseen malicious behaviors after exploiting known vulnerabilities, might be prohibitively difficult to detect by network or log mining. The goal of our work is to detect such exploits at the software level, and then (1) impede the attack by misdirecting the attacker, (2) lure the attacker to give defenders more time and information to trace and/or prosecute, (3) feed attackers disinformation to lower the effectiveness of current and future attacks, and (4) gather information about attacker gambits to identify and better protect confidential data against future attacks.

Real-world deployment. It would be worthwhile to evaluate the effects of honey-patching on web servers that are frequent victims of targeted, tailored attacks by resourceful adversaries. Over the long run, this can help us better understand the practical implications of honey-patching and the attacks they capture.

In addition to web servers, we consider our approach feasible for protecting web applications and other Linux networking applications in general. We would also like to test and deploy RedHerring on cloud infrastructures that support LXC to benefit from the cloud's dynamic scalability and performance characteristics.

8. RELATED WORK

Remote Exploitation. Remote, exploitable attacks are one of the biggest threats to IT-security, leading to exposure of sensitive information and high financial losses. While (zero-day) attacks exploiting undisclosed vulnerabilities are the most dangerous, attacks exploiting known vulnerabilities are most prevalent—public disclosure of a vulnerability usually heralds an increase of attacks against it by up to 5 orders of magnitude [9]. Most attacks are remote code injections against vulnerable network applications, and are automatically exploitable by malware without user interaction. Fritz et al. [23] survey the threat landscape of remote code injections and their evolution over the past five years.

Unfortunately, finding vulnerabilities that lead to remote exploits is becoming easier. ReDeBug [31] finds buggy code that has been copied from project to project. This occurs since programmers often reuse code, and patches are not applied to every version. The Automatic Exploit Generation (AEG) research challenge [6] involves automatically finding vulnerabilities and generating exploits by formalizing the notion of an exploit and analyzing source code. Security patches can also be used to automatically generate exploits, since they reveal details about the underlying vulnerabilities [12].

To help overcome this, patch execution consistency models, which guarantee that a patch is safe to apply if the tandem execution of patched and unpatched versions does not diverge, have been recommended as a basis for constructing honeypots that detect and redirect attacks [40]. Our work pursues this goal at the software level, where software exploit detection is easier and more reliable than at the network level. Software diversification has also been proposed as an efficient protection against patch-based attacks [15].

Honeypots for Attack Analysis. Honeypots are information systems resources conceived to attract, detect, and gather attack information. They are designed such that any interaction with a honeypot is likely to be malicious. Although the concept is not new [50], there has been growing interest in protection and countermeasure mechanisms using honeypots [7, 34, 37]. Honeypots traditionally employ virtualization to trap and investigate attacks [46, 61]. By leveraging VM monitors, honeypots adapt and seamlessly integrate into the network infrastructure [35], monitoring attacker activities within a compromised system [7, 19, 26, 37]. Nowadays, large *honeyfarms*, supporting on-demand loading of resources, enable large-scale defense scenarios [32, 56].

Shadow honeypots [1, 2] are a hybrid approach in which a front-end anomaly detection system forwards suspicious requests to a back-end instrumented copy of the target application, which validates the anomaly prediction and improves the anomaly detector's heuristics through feedback. Although the target and instrumented programs may share similar states for detection purposes, shadow honeypots make no effort to deceive attackers into thinking the attack was successful.

In contrast, OpenFire [10] uses a firewall-based approach to forward unwanted messages to decoy machines, making it appear that all ports are open and inducing attackers to target false services. Our work adopts an analogous strategy for software vulnerabilities, making it appear that vulnerabilities are unpatched and inducing attackers to target them.

Cloning for Security Purposes. Our work benefits from research advances on live cloning [51], which VM architectures are increasingly using for load balancing and fault-tolerance [36, 63]. In security contexts, VM live cloning can also be used to automate the creation of on-demand honeypots [8, 56]. For instance, dynamic honeypot extraction architectures [8] use a modified version of the Xen hypervisor to detect potential attacks based on analysis of request payload data, and delay their execution until a modified clone of the original system has been created. To fool and distract attackers, sensitive data is removed from the clone's file-system. However, no steps are taken to avoid leaking confidential information contained within the cloned process memory image, and the detection strategy is purely system-level, which cannot reliably detect the language-level exploits redirected by honey-patches.

9. CONCLUSION

This paper proposed, implemented, and evaluated honey-patching as a strategy for elegantly reformulating many vendor-supplied, source-level patches as equally secure honey-patches that raise attacker risk and uncertainty. A light-weight, resource-efficient, and fine-grained implementation approach based on live cloning transparently forks attacker connections to sandboxed decoy environments in which in-memory and file system secrets have been redacted or replaced with honey-data. Our implementation and evaluation for the Apache HTTP web server demonstrate that honey-patching can be realized for large-scale, performance-critical software with minimal overheads for legitimate users. If adopted on a wide scale, we conjecture that honey-patching could significantly impede certain attacker activities, such as vulnerability probing, and offers defenders a new, potent tool for attacker deception.

10. REFERENCES

[1] K. G. Anagnostakis, S. Sidiroglou, P. Akritidis, M. Polychronakis, A. D. Keromytis, and E. P. Markatos. Shadow honeypots. *Int. J. Computer and Network Security (IJCNS)*, 2(9):1–15, 2010.

[2] K. G. Anagnostakis, S. Sidiroglou, P. Akritidis, K. Xinidis, E. Markatos, and A. D. Keromytis. Detecting targeted attacks using shadow honeypots. In *Proc. USENIX Security Sym.*, 2005.

[3] J. Ansel, K. Arya, and G. Cooperman. DMTCP: Transparent checkpointing for cluster computations and the desktop. In *Proc. IEEE Int. Parallel and Distributed Processing Sym. (IPDPS)*, pages 1–12, 2009.

[4] Apache. Apache HTTP server project. http://httpd.apache.org, 2014.

[5] W. A. Arbaugh, W. L. Fithen, and J. McHugh. Windows of vulnerability: A case study analysis. *IEEE Computer*, 33(12), 2000.

[6] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. AEG: Automatic exploit generation. In *Proc. Network & Distributed System Security Sym. (NDSS)*, 2011.

[7] M. Beham, M. Vlad, and H. P. Reiser. Intrusion detection and honeypots in nested virtualization environments. In *Proc. IEEE/IFIP Int. Conf. Dependable Systems and Networks (DSN)*, pages 1–6, 2013.

[8] S. Biedermann, M. Mink, and S. Katzenbeisser. Fast dynamic extracted honeypots in cloud computing. In *Proc. ACM Cloud Computing Security Work. (CCSW)*, pages 13–18, 2012.

[9] L. Bilge and T. Dumitras. Before we knew it: An empirical study of zero-day attacks in the real world. In *Proc. ACM Conf. Computer and Communications Security (CCS)*, pages 833–844, 2012.

[10] K. Borders, L. Falk, and A. Prakash. OpenFire: Using deception to reduce network attacks. In *Proc. Int. Conf. Security and Privacy in Communications Networks (SecureComm)*, pages 224–233, 2007.

[11] B. M. Bowen, S. Hershkop, A. D. Keromytis, and S. J. Stolfo. Baiting inside attackers using decoy documents. In *Proc. Int. ICST Conf. Security and Privacy in Communication Networks (SecureComm)*, pages 51–70, 2009.

[12] D. Brumley, P. Poosankam, D. Song, and J. Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proc. IEEE Sym. Security & Privacy (S&P)*, pages 143–157, 2008.

[13] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proc. Sym. Networked Systems Design & Implementation (NSDI)*, volume 2, pages 273–286, 2005.

[14] Codenomicon. The Heartbleed bug. http://heartbleed.com, Apr. 2014.

[15] B. Coppens, B. D. Sutter, and K. D. Bosschere. Protecting your software updates. *IEEE Security & Privacy*, 11(2):47–54, 2013.

[16] J. Corbet. TCP Connection Repair. http://lwn.net/Articles/495304, 2012.

[17] S. Crane, P. Larsen, S. Brunthaler, and M. Franz. Booby trapping software. In *Proc. New Security Paradigms Work. (NSPW)*, pages 95–106, 2013.

[18] CRIU. Checkpoint/Restore In Userspace. http://criu.org, 2014.

[19] D. Dagon, X. Qin, G. Gu, W. Lee, J. Grizzard, J. Levine, and H. Owen. Honeystat: Local worm detection using honeypots. In *Proc. Int. Sym. Recent Advances in Intrusion Detection (RAID)*, pages 39–58, 2004.

[20] J. Duell. The design and implementation of Berkeley Lab's Linux checkpoint/restart. Technical Report LBNL-54941, U. California at Berkeley, 2002.

[21] J. Finkle. U.S. government failed to secure Obamacare site – experts. *Reuters*, Jan. 16, 2014.

[22] G. H. Friedman. Evaluation report: The Department of Energy's unclassified cyber security program. Technical Report DOE/IG-0897, U.S. Dept. of Energy, Oct. 2013.

[23] J. Fritz, C. Leita, and M. Polychronakis. Server-side code injection attacks: A historical perspective. In *Proc. Int. Sym. Research in Attacks, Intrusions and Defenses (RAID)*, pages 41–61, 2013.

[24] X. Fu, W. Yu, D. Cheng, X. Tan, and S. Graham. On recognizing virtual honeypots and countermeasures. In *Proc. IEEE Int. Sym. Dependable, Autonomic and Secure Computing (DASC)*, pages 211–218, 2006.

[25] Z. Gadot, M. Alon, L. Rozen, M. Atad, and Y. S. V. Shrivastava. Global application & network security report 2013. Technical report, Radware, 2014.

[26] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proc. Network & Distributed Systems Security Sym. (NDSS)*, pages 191–206, 2003.

[27] B. Gerofi, H. Fujita, and Y. Ishikawa. An efficient process live migration mechanism for load balanced distributed

virtual environments. In *Proc. IEEE Int. Conf. Cluster Computing (CLUSTER)*, pages 197–206, 2010.

[28] Google. Protocol Buffers. https://code.google.com/p/protobuf, 2014.

[29] Google. Web metrics. https://developers.google.com/speed/articles/web-metrics, 2014.

[30] T. Jackson, B. Salamat, A. Homescu, K. Manivannan, G. Wagner, A. Gal, S. Brunthaler, C. Wimmer, and M. Franz. Compiler-generated software diversity. In S. Jajodia, A. K. Ghosh, V. Swarup, C. Wang, and X. S. Wang, editors, *Moving Target Defense – Creating Asymmetric Uncertainty for Cyber Threats*, pages 77–98. Springer, 2011.

[31] J. Jang, A. Agrawal, and D. Brumley. ReDeBug: Finding unpatched code clones in entire OS distributions. In *Proc. IEEE Sym. Security & Privacy (S&P)*, pages 48–62, 2012.

[32] X. Jiang, D. Xu, and Y.-M. Wang. Collapsar: A VM-based honeyfarm and reverse honeyfarm architecture for network attack capture and detention. *J. Parallel and Distributed Computing – Special Issue on Security in Grid and Distributed Systems*, 66(9):1165–1180, 2006.

[33] W. Kandek. Year closing – December 2013 patch Tuesday. *Qualys: Laws of Vulnerabilities*, Dec. 2013.

[34] S. Kulkarni, M. Mutalik, P. Kulkarni, and T. Gupta. Honeydoop – a system for on-demand virtual high interaction honeypots. In *Proc. Int. Conf. for Internet Technology and Secured Transactions (ICITST)*, pages 743–747, 2012.

[35] I. Kuwatly, M. Sraj, Z. A. Masri, and H. Artail. A dynamic honeypot design for intrusion detection. In *Proc. IEEE/ACS Int. Conf. Pervasive Services (ICPS)*, pages 95–104, 2004.

[36] H. A. Lagar-Cavilla, J. A. Whitney, A. M. Scannell, P. Patchin, S. M. Rumble, E. de Lara, M. Brudno, and M. Satyanarayanan. SnowFlock: Rapid virtual machine cloning for cloud computing. In *Proc. ACM European Conf. Computer Systems (EuroSys)*, pages 1–12, 2009.

[37] T. K. Lengyel, J. Neumann, S. Maresca, B. D. Payne, and A. Kiayias. Virtual machine introspection in a hybrid honeypot architecture. In *Proc. USENIX Work. Cyber Security Experimentation and Test (CSET)*, 2012.

[38] Lighttpd. Lighttpd server project. http://www.lighttpd.net, 2014.

[39] LXC. Linux containers. http://linuxcontainers.org, 2014.

[40] M. Maurer and D. Brumley. Tachyon: Tandem execution for efficient live patch testing. In *Proc. USENIX Security Sym.*, pages 617–630, 2012.

[41] D. S. Milojičić, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou. Process migration. *ACM Computing Surveys*, 32(3):241–299, 2000.

[42] Netcraft. Are there really lots of vulnerable Apache web servers? http://news.netcraft.com/archives/2014/02/07, 2014.

[43] Nginx. Nginx server project. http://nginx.org, 2014.

[44] Ohloh. Apache HTTP server statistics. http://www.ohloh.net/p/apache, 2014.

[45] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable web server. In *Proc. Conf. USENIX Annual Technical Conference (ATEC)*, pages 15–15, 1999.

[46] N. Provos and T. Holz. *Virtual Honeypots: From Botnet Tracking to Intrusion Detection*. Addison-Wesley Professional, 2007.

[47] A. Sabelfeld and A. C. Myers. Language-based information flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, 2003.

[48] M. B. Salem and S. J. Stolfo. Decoy document deployment for effective masquerade attack detection. In *Proc. Int. Conf. Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 35–54, 2011.

[49] S. Souders. The performance golden rule. http://www.stevesouders.com/blog/2012/02/10/the-performance-golden-rule, Feb. 2012.

[50] L. Spitzner. *Honeypots: Tracking Hackers*. Addison-Wesley Longman, 2002.

[51] Y. Sun, Y. Luo, X. Wang, Z. Wang, B. Zhang, H. Chen, and X. Li. Fast live cloning of virtual machine based on Xen. In *Proc. IEEE Conf. High Performance Computing and Communications (HPCC)*, pages 392–399, 2009.

[52] The 111th United States Congress. An act entitled the patient protection and affordable care act. Public Law 111-148, 124 Stat. 119, Mar. 2010.

[53] The Economic Times. New technique Red Herring fights 'Heartbleed' virus. *The Times of India*, Apr. 15, 2014.

[54] J. Voris, N. Boggs, and S. J. Stolfo. Lost in translation: Improving decoy documents via automated translation. In *Proc. IEEE Sym. Security & Privacy Workshops (S&PW)*, pages 129–133, 2012.

[55] J. Voris, J. Jermyn, A. D. Keromytis, and S. J. Stolfo. Bait and snitch: Defending computer systems with decoys. In *Proc. Conf. Cyber Infrastructure Protection (CIP)*, 2012.

[56] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, and S. Savage. Scalability, fidelity, and containment in the Potemkin virtual honeyfarm. In *Proc. ACM Sym. Operating Systems Principles (SOSP)*, pages 148–162, 2005.

[57] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott. Proactive process-level live migration in HPC environments. In *Proc. ACM/IEEE Conf. Supercomputing*, 2008.

[58] J. Wang, X. Liu, and A. A. Chien. Empirical study of tolerating denial-of-service attacks with a proxy network. In *Proc. USENIX Security Sym.*, pages 51–64, 2005.

[59] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proc. ACM Conf. Computer and Communications Security (CCS)*, pages 157–168, 2012.

[60] A. Whitaker, R. S. Cox, M. Shaw, and S. D. Gribble. Constructing services with interposable virtual hardware. In *Proc. Sym. Networked Systems Design and Implementation (NSDI)*, pages 169–182, 2004.

[61] V. Yegneswaran, P. Barford, and D. Plonka. On the design and use of internet sinks for network abuse monitoring. In *Proc. Int. Sym. Recent Advances in Intrusion Detection (RAID)*, pages 146–165, 2004.

[62] J. Yuill, D. Denning, and F. Feer. Using deception to hide things from hackers: Processes, principles, and techniques. *J. Information Warfare*, 5(3):26–40, 2006.

[63] W. Zheng, R. Bianchini, G. J. Janakiraman, J. R. Santos, and Y. Turner. JustRunIt: Experiment-based management of virtualized data centers. In *Proc. USENIX Annual Technical Conf.*, 2009.