Improving Cybersecurity Hygiene through JIT Patching

Frederico Araujo*
IBM Research, Yorktown Heights
United States

Teryl Taylor
IBM Research, Yorktown Heights
United States

ABSTRACT

Vulnerability patch management remains one of the most complex issues facing modern enterprises; companies struggle to test and deploy new patches across their networks, often leaving myriad attack vectors vulnerable to exploits. This problem is exacerbated by enterprise server applications, which expose tremendous amounts of information about their security postures, greatly expediting attackers' reconnaissance incursions (e.g., knowledge gathering attacks). Unfortunately, current patching processes offer no insights into attacker activities, and prompt attack remediation is hindered by patch compatibility considerations and deployment cycles.

To reverse this asymmetry, a patch management model is proposed to facilitate the rapid injection of software patches into live, commodity applications without disruption of production workflows, and the transparent sandboxing of suspicious processes for counterreconnaissance and threat information gathering. Our techniques improve workload visibility and vulnerability management, and overcome perennial shortcomings of traditional patching methodologies, such as proneness to attacker fingerprinting, and the high cost of deployment. The approach enables a large variety of novel defense scenarios, including rapid security patch testing with prompt recovery from defective patches and the placement of exploit sensors inlined into production workloads. An implementation for six enterprise-grade server programs demonstrates that our approach is practical and incurs minimal runtime overheads. Moreover, four use cases are discussed, including a practical deployment on two public cloud environments.

CCS CONCEPTS

• **Security and privacy** → **Software and application security**; *Intrusion/anomaly detection and malware mitigation*; • **Software and its engineering** → *Software evolution.*

KEYWORDS

security patching, hot patching, cyber deception

ACM Reference Format:
Frederico Araujo and Teryl Taylor. 2020. Improving Cybersecurity Hygiene through JIT Patching. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3368089.3417056

*Both authors contributed equally to this research.

ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7043-1/20/11…$15.00
https://doi.org/10.1145/3368089.3417056

1 INTRODUCTION

Cyber attackers breach corporate networks using a myriad of techniques, with web application vulnerabilities corresponding to 25% of all exploitable attack vectors [26]. More disturbing is that these attacks can go unnoticed for hundreds of days [65], often resulting in the exfiltration of confidential company data and erosion of client trust. Organizations can no longer solely rely on perimeter defenses (e.g., firewalls, network intrusion detectors) to protect their IT environments; traditional network traffic monitoring and misuse detection systems are unable to keep up with evolving attacks, sustaining high error rates and being akin to searching for a needle in an extremely large haystack.

Companies also struggle to monitor, test, and deploy new security patches in their environments, affording adversaries the opportunity to perform low-risk reconnaissance operations. Attackers can quickly glean extremely valuable information on network topology, version, and patch level of running applications—all without the defender's knowledge. This favors attackers, who can wreak havoc by exploiting single vulnerabilities, while defenders are faced with the difficult task of protecting against all possible attacks.

To level this information asymmetry, this paper introduces *just-in-time* (JIT) patching as a methodology for agile security patch testing and exploit sensing. JIT patches fix vulnerabilities in live software applications while optionally embedding sensors along malicious application control flow paths to proactively signal attacks and collect attacker counterreconnaissance information. The new patching methodology enables the quick remediation of newly-discovered vulnerabilities, and allows security administrators to test new vulnerability patches for functionality and target environment compatibility—because JIT patches are reversible, application redeployment is not necessary between failed patch attempts. This allows security administrators to assess security patch risk, perform patch triage, and prioritize patch roll out.

Our patching technique is also effective against targeted attacks that use a probe payload for initial application reconnaissance and reserve a second payload for the actual attack. Towards this end, we describe a model in which JIT patching is used to enhance server applications with embedded deceptions that frustrate attacker reconnaissance and deceive adversaries into disclosing their attack payloads. With no downtime and minimal performance overhead compared to the original application, security administrators can leverage JIT patches to easily modify application responses to attempted attacks by injecting pieces of code written in high-level, easy-to-maintain source patches that are automatically compiled and applied to running applications. This creates high-accuracy *exploit sensors* and an active defense mechanism for immediate attack response. The sensors do not influence program execution under normal operation, and elicits a response that can be defined by the security analyst, which includes attack detection and misdirection via honey-patching [7], artificial software diversification for defending against return-oriented-programming attacks [20], and IDS crook-sourcing [5].

Unlike traditional honeypots, our work seeks to integrate deceptive capabilities into information systems with genuine production

value (e.g., servers and software that offer genuine services to legitimate users), placing our deceptive sensors along *real* attack paths. Such sensors dynamically patch existing functions in an application with attack detection and deception functionality. They are compiled into a bitcode using a patch synthesis process, and then injected into a running application or network service, where they are compiled further into machine code, and linked directly with the existing application. The original function is modified with a function trampoline [39], and subsequent calls to the original function will be directed to the new function. Furthermore, just as easily as deceptions can be deployed, they can be removed—leaving no trace in the application. We demonstrate our approach's feasibility on enterprise-grade, real-world legacy applications, including Apache HTTP, OpenSSH Server, bind, samba, and sendmail.

We also introduce a new *sandboxing technique* that can quickly transfer suspicious processes into a lightweight sandbox on demand. This enables the extraction of attack payloads for analysis and signature generation. These sandboxes provide added protection and constrain attacker's capabilities with no performance overhead.

This paper provides the following research contributions:

- A model for rapid, on-demand deployment of software patches in running legacy applications with no downtime and minimal performance overhead.
- A *just-in-time patching* mechanism (in-memory compilation of arbitrary C code for hot patching production applications) to enable live, custom patches with active responses to alert defenders and evade attacks.
- Four novel real-world application-level use cases, which are demonstrated and evaluated: *patch testing*, *version fluxing*, *hot honey-patching*, and *deceptive hot-hardening*.
- Implementations for *six* popular open-source server applications demonstrate that the approach is practical for performance-critical software with no disruption to legitimate workflows.

2 BACKGROUND & RELATED WORK

Application hot patching is the process of transparently updating a running program [68]—unlike *kernel hot patches* [8, 18, 43], which target live updates to the operating system, and are beyond the scope of this paper. In security contexts, hot patching can be instrumental to avoid downtime in production applications; however, it can leave programs in an unstable state. To address this limitation, Ramaswamy et al. [68] proposed Katana, an automated framework that employs a special type of ELF relocatable objects for supporting global synchronized data, which enable the verification of the results of hot patching ELF binaries. Payer et al. [64] used dynamic binary translation to demonstrate the feasibility of an update system for the Apache web server, patching 45 of 49 bugs at runtime with 7% overhead, and Huang et al. [38] introduced an autonomous hot patching (AHP) framework to automatically patch the binary code of live web-based applications. Brady et al. [12] describe a hot patching technique that injects shared libraries into a running process to replace existing functions.

There have also been several recent research efforts undertaken by the Software Engineering community in providing software maintenance tools that enable dynamic application updates [15, 16, 31, 55, 72]. For example, Chen et al. [15] implemented a system capable of dynamically updating three prevalent server applications: vsftpd, sshd, and httpd, with reported overheads of 5%. Neamtiu et al. [55] introduced a compiler-based approach to *dynamic software updating* (DSU) called Ginseng, in which a program is patched with new code and data while it runs. More recently, Rommel et al. [72] described a hot patching technique that does not pause the application by preparing a patched clone of the process's address space and migrating individual threads at predefined quiescence points. Makris and Bazzi [51] describe Upstare, a DSU approach that utilizes stack reconstruction to update active functions while maintaining both code and data representation consistency in multi-threaded applications. Similarly, on-stack replacement (OSR) [21] enables on-the-fly transitions between variants of a currently executed function. OSR is typically used by interpreters to JIT compile code sections (e.g., loops, functions) for better application performance.

The above approaches focus on software patching for software evolution, in which large feature updates requires strong guarantees. By contrast, we focus on application hot patching for testing and exploit sensing of vulnerabilities that typically involve smaller code changes and more code customization. As a result, we favored a simpler function-level patching approach that enables more dynamic code customization. Our approach therefore does not support function signature nor data replacement changes. We introduce JIT patching to enhance hot patching with the ability of injecting and performing in-memory compilation of reversible software exploit sensors in running legacy applications. A key aspect of our work is making the patches easily reversible and opaque to attackers.

Application-level deception. As cyber attacks become more sophisticated, there is an increasing need for better ways to detect and stop attackers. Cyber deception has garnered attention by both attackers and defenders as a weapon in the cyber battlefield. Rowe [73] describes cyber counter-deception as the use of planned deceptions to defend information systems against attacker deceptions. However, while such *second-order deceptions* [74] remain largely underutilized in cyber-defensive scenarios [36, 86], they are frequently used by attackers to search for evidence of honeypots [44, 52, 75, 88], avoid malware analysis [13, 78, 94], and conceal their presence and identity on exploited systems [9, 41]. In the virtualization domain, malware attacks often employ stealthy techniques to detect VM environments, within which they behave innocuously and opaquely while being analyzed by antivirus tools [17]. This underscores the need for counter-deception mechanisms that are capable of tricking and manipulating advanced attacker deceptions.

Honeypots are closely monitored information systems resources that are intended to be probed, attacked, or compromised, conceived purely to attract, detect, and gather attack information [67, 81] (c.f., [14] for a survey on honeypot research.). Traditional honeypots are usually classified according to the interaction level provided to potential attackers. *Low-interaction* honeypots present a façade of emulated services without full server functionality, with the intent of detecting unauthorized activity via easily deployed pseudo-services. *High-interaction* honeypots provide a relatively complete system with which attackers can interact, and are designed to capture detailed information on attacks. Despite their popularity [1, 14, 22, 32, 42, 66, 82, 85], both low- and high-interaction honeypots are often detectable by informed adversaries (e.g., due to the limited services they purvey, or because they exhibit traffic patterns and data substantially different than genuine services).

Application-level software deception differs foundationally from traditional honeypot technologies. Unlike honeypots, our work seeks to integrate deceptive capabilities into information systems with genuine production value (e.g., servers and software that offer genuine services to legitimate users). Fowler and Nesbit [29]

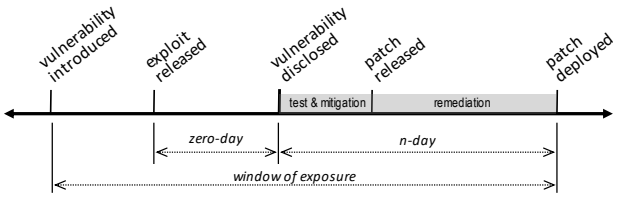

Figure 1: Vulnerability timeline showing a typical window of exposure. These events do not always occur in this order.

suggest six general principles for effective tactical deception in warfare, which prescribe that deceptions should (1) reinforce enemy expectations, (2) have realistic timing and duration, (3) be integrated with operations, (4) be coordinated with concealment of true intentions, (5) be tailored to contextual requirements, and (6) be imaginative and creative. These rules highlight limitations of current deception-based defenses. For example, conventional honeypots usually violate the third rule of integration as they are often deployed as *ad hoc*, stand-alone lures isolated from production servers. This makes them easily detectable by most advanced adversaries. They also assume that an adversary must scan the network in order to identify assets to attack. By contrast, application-level deceptions overcome these deficiencies because they can be deployed using real applications and real data.

Attacker session sandboxing. Unlike conventional application sandboxing approaches that are designed to protect the host environment against unintended or malicious code execution [10, 40, 46, 53, 92, 93], our approach benefits from recent advances in Linux namespaces [49], which wrap global system resources (e.g., users, process IDs, networking, filesystem) in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource. To enable on-demand application booby trapping, our sandboxing approach builds unprivileged and capability-constrained containers atop Linux namespaces, process control groups [48], and Linux capabilities [47], thus allowing the framework to keep the trusted computing base small and protect the application environment from attacker abuse.

3 APPROACH OVERVIEW

3.1 Vulnerability Management

Although there are no standardized protocols and formats for patch management, many patching guidelines have been published over the years. The National Institute of Standards and Technology (NIST)'s Special Publication (SP) 800-40 [57] and its subsequent revisions [58, 60] outline procedures for handling security patches and critical cybersecurity hygiene. Similarly, SP 800-184 [59] provides a guide for cybersecurity event recovery and recommendations for minimizing their impact on organizations. These can be leveraged to create playbooks for recovery from fast-spreading, severe cybersecurity incidents, such as NotPetya [33], EternalBlue [35], and Heartbleed [19]. Specifically, the National Cybersecurity Center of Excellence (NCCoE) describes a situational framework [80] for characterizing patching procedures, which include *routine patching* (for patches that are on a regular release cycle), *emergency patching* (for high-severity or actively exploited vulnerabilities), *emergency workaround* (for temporary mitigation prior to patch release), and *isolation of unpatched assets* (for systems that cannot be patched).

Just-in-time (JIT) patching improves vulnerability management by introducing new forms of emergency patching and security workarounds that facilitate patch testing and threat mitigation in enterprise environments. Figure 1 shows a typical vulnerability timeline. A *zero-day* attack typically exploits a vulnerability before its public disclosure. These exploits correspond to vulnerabilities that the security community is generally unaware of. Conversely, a *n-day* attack denotes an exploit of a known vulnerability. These exploits target unpatched software for which permanent fixes are slowed by patching considerations and operational requirements.

When harnessed as part of a rigorous security program, JIT patching can help organizations better defend against n-day attacks through (1) fail-safe patch testing and threat sensing capabilities, and (2) deployment prioritization of new patches with reduced impact of potential patch failures.

3.2 JIT Patching

There is typically a large time window between when a new application vulnerability is found until a full scale patch is deployed. During this time, attackers have free reign to exploit systems unchecked. Our approach provides a layered security mechanism to manage this vulnerability gap. By incorporating JIT patching in the enterprise *pre-patch* routine (i.e., prior to rolling out new versions of an application), the framework can quickly respond to vulnerability disclosures by immediately addressing attack vectors while gathering potentially unknown attack payloads, even before proof-of-concept attacks become available. As a result, security administrators can prioritize patch compatibility testing, and use any captured payloads to create network signatures and protect unpatched assets [5]. The new patching methodology also helps defenders overcome key limitations of honeypots and perimeter-based defenses by embedding programmable deceptions and sensors along genuine attack paths to gather contextual information, which is useful in tracking, confusing, and denying potential attackers.

To effectively realize this security patching methodology, many challenges must be overcome. Chief among them is that JIT patches must be transparent to normal users, preserving the application's normal behavior, while responding smartly in the face of an attack. Second, the deployment model must be able to respond rapidly to the changing threat landscape by injecting sensors anywhere in the network, on demand. Furthermore, the embedded sensors must be easily removable without a trace so as not to tip-off attackers. Finally, the framework must not impair the function or performance for legitimate workloads, nor create timing channels to alert the attacker of application deception.

To cope with the challenges of rapidly deploying application sensors, we propose Insider, a framework for rapid, on-demand application JIT patching. Figure 2 depicts the process of injecting a patch into a running application (steps ①–③) followed by a response triggered by that patch (steps ④–⑤). The process consists of an offline step, and four online steps. In step ①, a patched function (or functions) is written to replace an original function (or functions) in the running application. The patch is compiled into bitcode, and symbols are extracted from a copy of the app binary in an offline process. Once the patch is synthesized, it is deployed to the target application and injected into the application's memory space (step ②). The patch is then compiled on a separate execution thread inside the process into native code, and linked against the global symbols of the application using the symbol mappings from the patch synthesis phase. Next, the process is briefly paused to

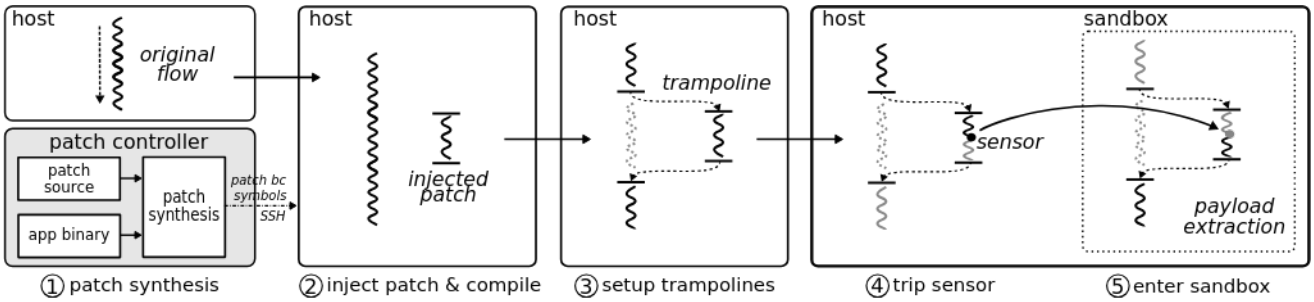

Figure 2: The Insider framework: Process of injecting a JIT patch into a running application (steps ①–③) causing (a) no change to benign execution flows (step ③) and (b) threads optionally tripping over the exploit sensor (step ④) to be moved into a sandbox for suspicious payload extraction (step ⑤).

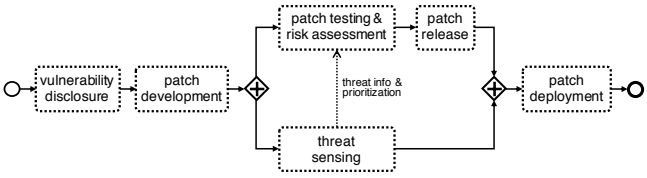

Figure 3: Vulnerability management enhanced with JIT patching for rapid patch testing and threat mitigation.

insert a trampoline from the function that is replaced with the patch as shown in step ③. When an attack payload triggers a patched function (step ④), the application elicits a user-defined response. The response can be a *passive* action, such as terminating the connection, collecting the attack payload, and notifying a security analyst, or an *active* action, such as transparently moving the current execution thread of the application into a decoy sandbox (step ⑤) for further execution and in-depth analysis, providing disinformation, and temporarily activating rules to block an IP address. Note that data structure replacements is outside the scope of our work.

3.3 Patch Testing and Threat Mitigation

Figure 3 shows a workflow outlining the sub-processes involved in a vulnerability management routine augmented with JIT patching. Upon discovery of a new vulnerability, a candidate patch is developed, or accepted from a third-party vendor. This initiates two complementary pipelines that leverage Insider to (1) test patch compatibility with the operating environment prior to final patch release, and (2) live patch strategic assets with embedded threat sensors to mitigate the vulnerability and collect threat information for patch prioritization and permanent remediation. Next, we describe use cases that are non-trivial to achieve in commodity, legacy server applications. Design and implementation are revisited in §7.

Security patch testing and risk assessment. Security patch testing is a critical challenge facing enterprises today. Companies need to prioritize patch testing and installation based on many factors, including severity, exposure, environmental compatibility, service availability, and resources. JIT patching enables patches to be rolled out quickly for testing, and with reduced impact of patch defects. Patches can also be repurposed as exploit sensors to gather statistics on concrete vulnerability exposure to attacks. This helps security analysts optimize patch management and deployments, and improve security risk assessment (§7.2).

Attacker reconnaissance is a key phase in targeted attacks against enterprise networks. Advanced adversaries probe their victim's systems to enumerate potential vulnerabilities before launching offensive campaigns. In web contexts, identifying the web server version is accomplished by probing the server, for exposed server banner information, distinctive error pages, and implementation details (e.g., HTTP header ordering, handling of malformed requests). To counter such *fingerprinting* attempts to gain system information, *version fluxing* [20] has been conceptualized as a program diversification strategy that randomly modulates the behavior of different versions of a target application. This adds noise to attacker probes, frustrating version reconnaissance attempts.

To demonstrate Insider's utility in protecting against application fingerprinting, we extend version fluxing—originally rooted on compiler-based instrumentation and software obfuscation—with JIT patches that conceal or alter collected attacker intelligence. Our implementation for the Apache HTTP web server (§7.3) highlights our approach's ability to quickly manipulate attack responses.

Security patching. When a software security vulnerability is discovered, the conventional defender reaction is to quickly patch the software to fix the problem. This standard reaction can backfire if the patch has the side-effect of disclosing other exploitable weaknesses in the defender's network. Unfortunately, such backfires are common; patches often allow adversaries to infer which systems have been patched, and which are vulnerable. *Honey-patching* [6, 7] introduces deception into security patching. In response to malicious inputs, honey-patched applications clone the attacker session onto an isolated decoy environment, which impersonates an unpatched, vulnerable version of the software.

Insider's dynamic patch injection facility enables rapid honey-patching of newly-discovered threats by dynamically retrofitting the *running* server with an embedded sandbox that monitors and disinforms attackers (§7.4). Such *hot-honeypatching* is useful in contexts where immediate and comprehensive patch deployment—the conventional defense—is infeasible or impractical [25].

Configuration hardening. Improper application configuration settings have the potential to jeopardize the security posture of an organization. To alleviate this problem, *hot-hardening* introduces an agent that continuously selects and deploys new configuration settings to enhance application security. For example, a hot-hardening agent may discover that an OpenSSH server was mistakenly configured to allow password authentication—in violation of organization policy—and patch the server. Although hot-hardening may be seen

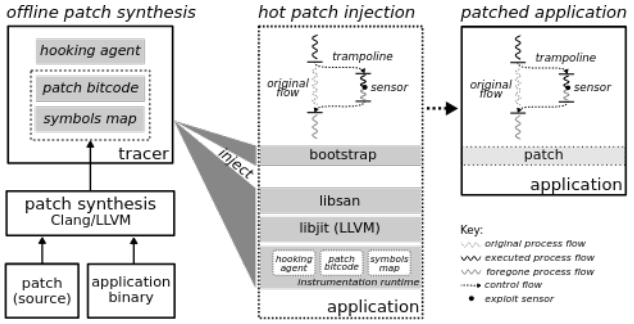

Figure 4: Patch synthesis and injection, showing final application address space (right) with auxiliary runtime instrumentation and libraries sanitized from the process address space after JIT patch compilation and injection.

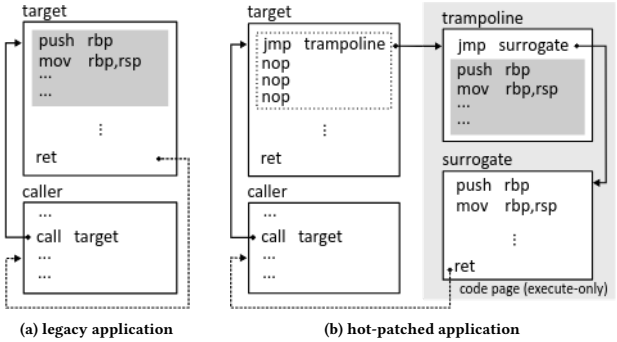

Figure 5: Function trampolining. (a) Unmodified legacy code. (b) Flow diverted to surrogate function via trampoline.

as a form of proactive defense, it fails to provide defenders with any insights on how such erroneous configurations are abused.

Alternatively, we propose *deceptive hot-hardening* as a new use case for configuration hardening, employing Insider to make an application appear misconfigured in order to gather intelligence on attackers who attempt to exploit the weakness. We showcase this new approach by injecting configuration patches into live OpenSSH daemons deployed on a cloud environment (§7.5).

4 ON-DEMAND PATCH INJECTION

This section details Insider's on-demand live patch injection mechanism, which comprises three major steps: (1) patch synthesis, (2) patch injection with in-memory patch compilation, and (3) function replacement via trampolining. Each step is discussed in more detail in the following subsections.

4.1 Offline Patch Synthesis

Patch synthesis takes place offline and creates the intermediate bitcode used to patch the target process, as shown in Figure 4. The synthesis step takes as inputs the patch source code containing new implementations of functions to be replaced/added and a metadata object. This metadata is maintained as a separate file by administrators and describes the patched functions in terms of name, argument types, and return type. It also contains a description of patch dependencies, such as external libraries and linking type (static or dynamic) to inform the JIT runtime of the appropriate strategy to compute symbol addresses for linking JIT-compiled patch functions to the target process. With this information, the JIT compiler compiles the patch source code to a bitcode representation—a typed, static-single-assignment intermediate representation—and extracts a symbol map from the symbol tables of a copy of the target binary. Finally, the *tracer* component packs the patch bitcode, the symbols map, and a generic hooking agent into a shared object (.so). A hooking agent is a binary object tasked with hooking capabilities, and injects a bootstrap code routine that launches an ephemeral processing thread inside the process.

4.2 Dynamic Patch Injection

Once patch synthesis completes, the tracer injects the JIT (`libjit`) and instrumentation (`libsan`) runtime shared objects into the target process address space using the `ptrace` system call, and patches the target process (see the transient program address space shown in Figure 4). Specifically, the tracer component determines the process IDs associated with the target application. For each process ID, the tracer attaches to the target process and saves the state of the process' registers. The tracer then maps the instrumentation runtime object and a bootstrap into the target's address space. The target process is detoured into executing the bootstrap, which in turn launches a separate thread. The thread compiles the patch down to binary code and executes functions that apply the patch to the application (see §4.3). After the thread is launched, the tracer restores the target's registers to their original state. Finally, the tracer detaches from the target process, and the process resumes.

JIT compilation and linking. As mentioned above, the hooking agent's thread performs the in-memory compilation and linking of patch code, and sets up trampolines to replace target functions with patched ones. To achieve this, the hooking agent initializes a JIT compiler and computes the relocated addresses of dynamic symbols. The algorithm first identifies all the libraries linked in the target process. If an application or library is position independent, then the algorithm calculates its base address in the running process. Finally, the base address is added as an offset to each symbol in the symbol map that was created during the offline patch synthesis phase. With symbol mapping completed, the JIT compiler compiles the bitcode to machine code, which is stored in memory. The patch is then linked to the target process using the updated symbol map and the JIT compiler and auxiliary patching code are removed from the process address space in order to minimize the trusted computing base (patched process state in Figure 4).

Function trampolining. To support function replacement, the hooking agent writes a jump instruction (followed by an aligning `nop` sled) at the entry point of each target function that points to the corresponding patched function using an intermediate trampoline [39]. Figure 5 shows the before- (a) and after- (b) shots of replacing a target function with a JIT compiled surrogate function using a trampoline. Note that the overwritten code section of each target function is appended to the trampoline, allowing the hooking agent to revert the trampoline as desired.

Remote Injection. The patch controller (Figure 2 step ①) deploys patches to network hosts over SSH using public key authentication. The controller pushes a tarball including patch injection scripts, dependencies, patch bitcode, configurations and target application symbols over to the host. It then runs the patch injection script,

which compiles the patch bitcode into the target application and sets up the trampoline. Once completed, the JIT compiler and supporting scripts are removed from the system, leaving the compiled patch in place, as shown in the final process address space in Figure 4.

Process de-instrumentation. During the unpatching process, the tracer does the reverse of the injection process. First, the tracer determines the process IDs associated with the target application. Then, for each process ID, the tracer attaches to the target process and saves the state of the target process' registers. It then triggers a hooking agent cleanup function in a separate thread to perform the patch unhooking and JIT runtime cleanup. Once the thread is completed, it signals the tracer to unmap the instrumentation runtime object from the target's address space, deletes the bootstrap, and restores the target's registers. Finally, the tracer detaches from the target process, and the target process resumes execution.

To revert the code trampolines, the hooking agent thread reverts all previously created trampolines by replacing the indirect jumps (and associated `nop` sleds) located at the entry point of each target function with its original instructions (previously stored adjacent to the trampoline code). Finally, the hooking agent unmaps the address ranges corresponding to the patch code. Once removed, there is no trace of the patch to tip-off an attacker.

4.3 Patching API

To facilitate application patching, we define a small API that enables users of our framework to define and execute *patch injection scripts* on the hooking agent's thread within the application address space:

- `compile(`*bc*, *symbols*`)`: compile bitcode *bc* in the target's address space
- `run(`*fn*, *args*`)`: run a function *fn* in the target's address space
- `replace(`*fn*, *fn′*`)`: replace *fn* with *fn′* in the target process
- `sandbox()`: clone the target process into process sandbox

Function `compile` is used to compile and link patches inside the target process, while function `run` executes the patched function in the target's address space—this is often used to change configurations inside the process. Function replacement is done by calling `replace`, and a process is moved into a sandbox using `sandbox`.

Our current implementation embeds these user-defined scripts into the instrumentation runtime library, which uses a set of C-based stubs to interpret and execute them. Example scripts are shown in §7 as part of our use cases. Note that once a script has completed, the instrumentation runtime and JIT compiler libraries are removed from the application so that attackers cannot use these libraries to instrument their own attacks.

5 UNPRIVILEGED SANDBOXING

Once a sensor is tripped, Insider can either stop the attack and emit an alert, or fork the attacker into an unprivileged sandbox for further analysis. This enables security analysts to gain further insight into the attack (such as what malware is downloaded), which can be used to search for similar attacks on the network, or provide threat intelligence to prioritize patch deployment.

5.1 Decoy Sandbox Architecture

The architecture is based on operating system container technology and uses *namespaces* [49] to control memory, network, filesystem, user privileges, and process isolation from the global system. The process sandbox is built on top of a customizable *decoy filesystem* that contains all the system files commensurate with a real OS filesystem, but is isolated from the host filesystem. A *controller* creates the sandbox and monitors its activities transparently from the attacker process. Network packets and system traces are collected during the lifetime of the sandbox and stored remotely for analysis.

When an attacker trips a sensor using our patching framework, the attacked process clones itself via a `fork` system call into the sandbox, while the original process is recycled for further operation. The clone enters the sandbox namespaces by making a number of `setns` system calls. Given the structure of namespaces, only a process's child can enter the PID namespace; therefore, the copy of the attacked process must fork again, leaving the process and the attacker inside the sandbox. The sandbox can be terminated by a predefined timeout or at the discretion of the security administrator.

Table 1: Benchmarked applications

Application	Version	Architecture	Benchmark*
Apache (HTTP) [4]	2.4.6	multi-processed	`ab` [4]
nginx (HTTP) [56]	1.6.0	multi-threaded	`ab` [4]
Sendmail (SMTP) [77]	8.15.2	multi-processed	custom
bind (DNS) [11]	9.10.3	multi-processed	`DNSperf` [23]
vsftpd (FTP) [87]	3.0.2	multi-processed	`ftpbench` [30]
samba (smb) [76]	4.6.3	multi-processed	custom

*Load of 5k requests in 10 concurrent threads.

6 IMPLEMENTATION

Our prototype implementation consists of 4,957 lines of C/C++, Python, and JavaScript code. JIT Patches are crafted in C/C++, and compiled to bitcode using LLVM [50]. We use the `ptrace` system call to inject the patch, and natively compile the code using a custom library that wraps the LLVM Compiler MCJIT API. To create the function trampolines and hooking agent, we leverage Frida's API[69] while Python scripts monitor the injected processes. The sandbox is built on top of a Docker container [24] and utilizes control groups [48] for resource control, and Linux capabilities [49] to limit permissions of sandboxed processes. We deployed a standard Ubuntu image for the decoy filesystem, and monitor sandbox activity using `sysdig` [84]. System traces and alerts are collected in a `fluentd` [28] wrapper, and can be forwarded to various security data lakes and intrusion prevention systems. The patch controller is implemented using python scripts and Ansible playbooks [2].

7 EVALUATION

To demonstrate the efficacy of our approach, we conducted performance benchmarks on *six* popular server applications, and developed *four* use cases to showcase Insider's flexibility in enhancing commodity applications with active security-aware patches: *Version fluxing*, using Apache HTTP as the target application, *hot honeypatching* using Samba, and *deceptive hot-hardening*, using OpenSSH sshd. To further validate the practical aspects of our framework, we deployed the security-aware sshd server on two public clouds, and monitored the deployment for 10 days.

Our findings show that our framework can cope with a broad range of application architectures, adding minimal overhead to the applications we tested (−2% to 1%), even under strenuous attack workloads and large injected patches (1000+ LOC).

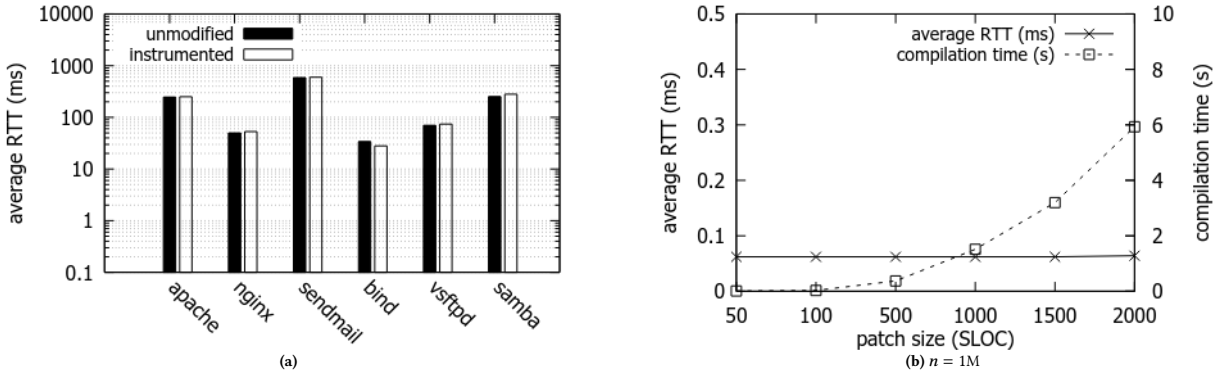

Figure 6: Performance benchmarks. (a) Runtime overhead: Effect of function trampolining on application round-trip time (10 concurrent user threads). (b) JIT compilation overhead: Effect of in-process compilation on performance when varying patch size (Nginx server under constant load).

7.1 Performance Benchmarks

We evaluated our framework on six legacy server applications including two web servers (Apache HTTP, nginx), a DNS server (bind), an email server (sendmail), a file sharing application (samba), and an FTP server (vsftpd). Table 1 summarizes the benchmark tools used to test each application, server versions and their multi-processing architectures. These applications were chosen due to their diversity and wide popularity. They also represent a diverse array of process architectures (see Table 1) that create some interesting challenges for the framework deployment, as discussed in our use cases.

Experimental setup. Experiments in this section were conducted in a virtual machine (VM) running Ubuntu 16.04, with 8 GB of memory, and a 2.50 GHz dual-core Intel i7-4780HQ CPU. Each tested application was installed unmodified and using default configurations. To measure web server response times consistently across benchmarks, a default index page was downloaded from each web server using `ab` 5k times, and we used the first 5k domain names from the example query domain set provided with `DNSperf` to benchmark bind. In addition, to evaluate sendmail, a custom benchmark was written in `python` using the built-in `smtplib` library. The tool selects from 36 email bodies (each 10 kB in size) that are sent to the server. We used `ftpbench` [30] to request a list of files in the home directory of `vsftpd` during the evaluation. Finally, for samba we created a random 64KB file stored on the samba share using `dd`, and downloaded the file 5k times using the `pysmb` library.

Each experiment was run three times and all memory, disk, and application caches were cleared between runs. All benchmarks and applications were run locally on the virtual machine.

Runtime overhead. In this experiment, we measured the impact of Insider's function trampolining facility on application performance. Towards this goal, for each server application we chose a function satisfying the following criteria: it implements core functionality and is called once per client request (e.g., Apache's `ap_process_request` is called once per HTTP request). For each of these core functions, we crafted a patch mirroring the implementation of the original function, and injected it into the corresponding tested application using Insider to replace the original function.

Figure 6a shows the average round-trip time (RTT) measured for each application, when under a load of 5k client requests from 10 concurrent user threads. The chart shows the performance of both patched and unpatched applications. Our measurements show overheads of 0.63%, 6%, 1.5%, −20%, 5.7%, 10% for Apache, nginx, sendmail, bind, vsftpd, and samba respectively. Since server computation accounts for only about 10% in practice [79], this corresponds to observable overheads of about 0.063%, 0.6%, 0.15%, −2%, and 1% (respectively). Indeed, in certain cases injecting the patch actually improves performance (bind), due to the optimizations that the JIT compiler can add during the compilation of large patches.

JIT compilation overhead. In this benchmark, we measured the impact of compilation time on application performance. We generated patches with 50–2k lines of code. Each patch bitcode uses a combination of `add` instructions to fill the desired length (no compiler optimization is performed in translation of source code to bitcode). The patches were then injected and compiled inside a running instance of nginx, while the web server was under a steady load of 16k HTTP requests per second. We chose nginx for this experiment because it is an event-driven web server that supports a single-processing model, therefore simplifying our evaluation.

Figure 6b shows compilation time for each patch size along with average server response times. The patch compilation time is almost linear with its size, while average RTT remains constant for all patches indicating that compilation does not interfere with the performance of the application. This favorable performance characteristic stems from Insider's design, which performs JIT compilation in a separate, low-priority thread of the live application, thus amortizing compilation costs over time.

Attacker sandboxing overhead. To assess the overhead of dropping a forked process into a sandbox during an attack, we honey-patched an Apache webserver that would drop a worker process into a sandbox during a Shellshock attack. The Shellshock GNU Bash remote command execution vulnerability (CVE-2014-6271) [61] was one of the most severe vulnerabilities in recent history, affecting millions of then-deployed web servers and other Internet-connected devices. We omit the details of the patch due to space constraints.

Figure 7 shows the average round-trip-time when varying the number of concurrent requests against Apache for (1) an unpatched version (baseline), (2) a version hot-honey-patched with Shellshock but no attack traffic, and (3) a version hot-honey-patched with Shellshock, but all requests are attacks and are forced into a sandbox.

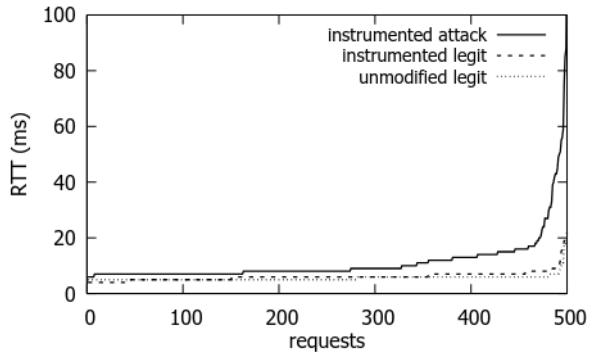

Figure 7: Sandboxing overhead: Effect of sandboxing (Apache server unpatched with legitimate requests [dotted], Shellshock honey-patched with legitimate [dashed] and Shellshock attack [solid] requests).

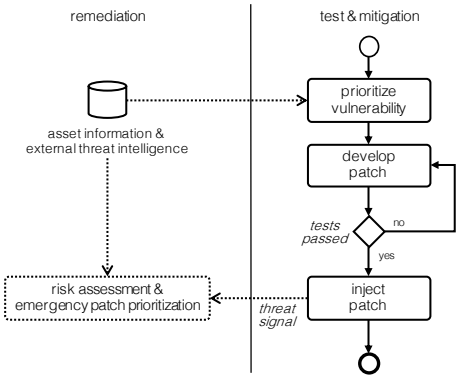

Figure 8: Simplified patch testing and risk assessment workflow leveraging JIT-patched application sensors.

Listing 1: Apache HTTP version fluxing hot patch

```
1 const char*_ap_get_server_banner(void) {
2     return generate_server_banner(version);
3 }
4 void _ap_send_error_response(request_rec *r, ...) {
5     error_response_factory(version);
6 }
```

Listing 2: Apache HTTP version fluxing script

```
1 compile(bc, smap)
2 replace(ap_get_server_banner, _ap_get_server_banner)
3 replace(ap_send_error_response, _ap_send_error_response)
```

The benchmarks were done using Apache's `ab` tool, single threaded (c=1). Results show that patched and unpatched versions perform similarly as the number of concurrent are increased. Furthermore, continuously entering the sandbox, under constant Shellshock attack adds almost no overhead up to 300 concurrent attacks. Note that as the number of concurrent requests increases, so does the the number of Apache worker processes, which indicates that the additional overhead observed after 300 concurrent attacks is dominated by the constant context switching among worker processes.

7.2 Use Case: Patch Testing & Prioritization

An important challenge in vulnerability patch management is prioritizing which patches to deploy into an enterprise. Each new patch requires the re-installation of an application, and a testing phase. As a result, roll out is slow, requiring enterprises to prioritize patch deployment. To improve this process, Figure 8 outlines a simplified workflow through which patches can be rapidly deployed onto a subset of the existing infrastructure while signaling security analysts when attempted exploits are levied against applications. By recording attempted attacks, analysts can determine (1) which applications are at greater risk of attack, and (2) which vulnerabilities are being exploited. Such metrics, combined with vulnerability score information [62] and open-source threat intelligence [63], enable enterprises to better assess security risk and prioritize patch deployment coupled with operational risk obtained from the patch sensors. Rapid deployment reduces the overhead for patch testing, enabling faster testing phases, with more reliable outcomes and reduced impact of patch failures (due to the ability of quickly reversing deployed patches).

7.3 Use Case: Version Fluxing

Version fluxing [20] enables an application to impersonate other applications to avoid attacker fingerprinting. In this use case, we show how a running Apache [4] web server instance can be modified to impersonate another web server (e.g., nginx [56], Microsoft IIS [54]) by modifying its response banner, and error response. Apache is a multi-process application with one root process that manages a pool of worker processes that handle client requests. Our framework automatically patches each worker and root process.

Apache handles error responses using the function `ap_send_error_response` and populates the response banner using `ap_get_server_banner`. Our goal is to leverage Insider to override these functions in a running Apache server in order to make its responses look like nginx responses. Listing 1 shows part of the patch to implement version fluxing. It customizes functions `ap_send_error_response` and `ap_get_server_banner` to generate an nginx banner and error responses. Listing 2 shows the script that is injected into the worker processes to install the patches. First, the patch bitcode (bc) is compiled into machine code using the supplied symbols (smap), then the functions are replaced to make the Apache server impersonate nginx.

7.4 Use Case: Honey-Patching

In this use case, we employ Insider to enable *hot honey-patching* of a live samba server, to deceive attacks targeting the SambaCry [34] exploit (CVE-2017-7494). The SambaCry remote code execution vulnerability was a high-impact vulnerability discovered in 2017 affecting millions of Linux servers, raspberry pi's, and Internet of Things devices. The vulnerability allows a remote samba client to upload an arbitrary shared library, escalate privileges, and execute a compiled function, enabling the attacker to take over the system. The exploit takes advantage of a feature in samba that allows certain modules to be executed remotely through named pipes.

Hot Honey-patching Samba. Our target server is an unmodified samba server configured to allow anonymous clients write access.

Listing 3: Hot honey-patch for SambaCry vulnerability

```
1 int _p_is_known_pipename(const char *pipename, struct
    ndr_syntax_id *syntax) {
2     if (strchr(pipename, '/')) {   // check for arbitrary library
3         sandbox()    // fork process to sandbox
4     }
5     return is_known_pipename(pipename, syntax)
6 }
```

Listing 4: OpenSSH server patches for changing the server configuration and redirecting an invalid user.

```
1 void update_options (char* new_config) {
2     // load configuration from new configuration file
3     load_server_config(new_config, &cfg);
4     parse_server_config(&options, new_config, &cfg, NULL);
5 }
6 int _p_auth_password(Authctxt*user_ctxt, const char*password) {

7     ...
8     if (!is_valid(user_ctxt)) {// user in db, wrong authentication
9         user_ctxt = get_decoy_user();// switch to decoy user
10        options.forced_cmd = "redirectUser.sh " + user_ctxt
            ;// redirect to decoy
11    }
12    return sshpam_auth_passwd(user_ctxt);
13 }
```

Samba is a multiple process application with a root process that forks worker processes to serve new incoming connections. As a result, we can hot patch all processes by simply patching the root.

The hot honey-patch is a re-implementation of the method `is_known_pipename` as shown in Listing 3 (the injection script is trivial and omitted for brevity). The patch augments the original function with a conditional statement (lines 2-4), which ensures that the samba server does not load an arbitrary shared library. If an attacker attempts to exploit the vulnerability, the samba worker process handling the request forks itself into a decoy sandbox (line 3), which records that attacker's actions as described in Section 5.

7.5 Use Case: Deceptive Hot-Hardening

Application hardening is the process of changing an application's default configuration in order to improve security. In this use case, we describe how to use Insider to dynamically change the configurations of an OpenSSH server on a per client basis in order to deceive an adversary into believing there is a weak configuration setting. We term this process *deceptive hot hardening*. An example of deceptive hot hardening is allowing a cipher negotiation between an ssh client and server to determine whether the client will choose a weak cipher, and then recording that intelligence. Another example is supplying password prompts in an OpenSSH server that only accepts key-based authentication. We can offer password prompts for those trying to use password-based authentication in order to ascertain whether an attacker has stolen an employee's credentials and are using them incorrectly. These credentials can be stored, and the attacker transparently redirected to an OpenSSH decoy server.

Listing 5: OpenSSH Server scripts

```
1 compile(bc, smap)
2 if (pid == ROOT) {
3     run(update_options, ["/etc/ssh/ssh_config_dhh"])
4 } else {
5     replace(auth_password, _p_auth_password)
6 }
```

OpenSSH is one of the most important applications in an enterprise network enabling users to remotely access enterprise resources. OpenSSH servers are often the targets of attacks making them ideal for injection with active deceptions using our framework. The OpenSSH server uses a multi-process architecture similar to Apache with some important distinctions. Like Apache, a root process is always running, and accepts new SSH connections. With a new incoming connection, the root process spawns a process to handle the connection. However, unlike Apache, this process is not forked from the root, but rather executed using an `exec` system call. This newly created process (known as the privilege separation monitor) forks off a process (called the user process) that interacts with the client. The user process communicates with the privilege separation monitor in order to authenticate the user. This architecture is an example of a privilege separation model, and is beneficial because even if the attacker compromises the user process, he has no access to the privileged authentication APIs.

Patching OpenSSH. JIT Patching an OpenSSH server is more complicated because of the three process architecture, and the fact that the root process launches, rather than forks, a privilege separation monitor. To deal with this unique architecture, we take the strategy of injecting code to the root process that changes OpenSSH Server configurations on the fly (and specific to each privileges process spawned). By changing configurations, we force the privilege separation monitor to execute with our framework already attached. From the privilege separation monitor, we can change the authentication properties of the application, collect forensic information about the user, force the user into particular actions, and attach to the forked user process.

As mentioned above, we can alter the configurations of the OpenSSH server to deceive attackers and collect information. Configurations are altered by injecting a method into the binary, which forces the global `config` object in OpenSSH to load a new configuration file (see `update_options` in Listing 4). The function is then run inside the application using Insider's `run` API (Listing 5 lines 2-4). Note, we use this same technique to force the root process to load a privilege separation monitor with our framework attached.

Along with changing configurations to collect information about would-be attackers, we can add in active responses. Take for example, the case of an OpenSSH server that only accepts public/private key authentication. We can change the configuration to allow negotiation of a password, and then monitor users trying to login to the server with passwords. Indeed, in our deployment environment, we see thousands of SSH login attempts per day. Most are brute force attacks using popular user names and default passwords for IOT devices, or root password dictionary attacks. In amongst all the noise, there are cases where legitimate enterprise user ids are attempting to login using passwords to SSH servers that only accept key authentication. For these instances, we can modify OpenSSH server's authentication code to allow user ids, that appear in the

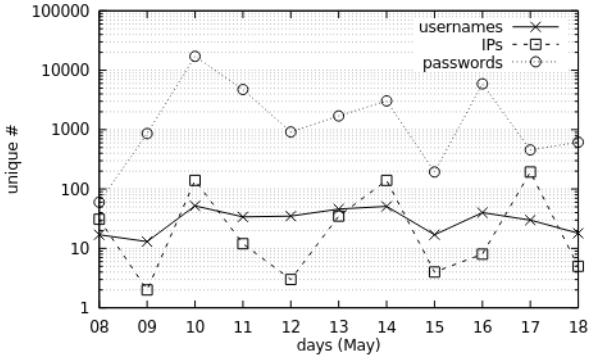

Figure 9: Distribution of unique usernames, IPs, and passwords collected in a patched `sshd` server instance

Linux user database, to be redirected to a decoy sandbox. Listing 4 (`_p_auth_password`) shows a patch that does such a redirection when injected into the privilege separation monitor. If the user does appear in the database, but is using a password to authenticate, he is redirected to a decoy sandbox on a separate system using an OpenSSH server forced command (lines 8-11) . In other words, if the user is using the wrong authentication method, OpenSSH automatically calls a script that creates the user id on a decoy, and forwards the SSH session to that decoy.

7.6 Deployment

We patched production instances of OpenSSH Server using Insider on two public clouds (anonymized). The patch modifies the configuration to allow password authentication and records the user name and passwords of those accessing the server. Since we are the only legitimate users of the system, all other attempted logins are deemed malicious. Looking at our SoftLayer instance, over a ten-day period from May 8–18 there were 135,872 login attempts from 402 unique IP addresses mainly located in China. The framework handled the request load with no crashes underscoring the stability of Insider.

Figure 9 shows the number of new user names, IP addresses, and passwords seen per day. It took a day for adversaries to find the server, and attack traffic was bursty with single IP addresses conducting dictionary attacks using the root user name. Indeed, 133,000 of the requests were with the root user name. More interesting, there were 82 user names that had between 1-10 login attempts each. We are currently trying to gain access to our enterprise user id list to verify how many user ids are from legitimate corporate accounts. With the list in hand, we hope to use our redirection code to study attackers actions in the wild.

8 DISCUSSION

Source code availability. Our current prototype implementation targets open-source software, but we do not see this as a technical limitation. In the near future, we plan to investigate deployment models that will enable the utilization of Insider with closed-source applications. More specifically, we want to explore ways to eliminate the need for symbols from a non-stripped binary to do symbol mapping in a running application. We envision Insider as a framework affording software vendors to build their own custom deceptive sensors. This capability can be leveraged by users to add their own code in order to integrate with other systems.

Compatibility. Insider's symbol address resolution algorithm supports all four address space layout randomization techniques (i.e., stack, exec, lib, and brk) available in mainstream Linux. Position independent code is also supported. These techniques are supported because patches are compiled and resolved while the application is running. We have not tested our approach with any finer grain randomization algorithms [91] as there are currently no production-ready techniques supplied with Linux.

For our initial prototype, we assume that all patched functions have the same argument list and return values as their corresponding originals. Our work focuses on changes to program control flow. Adding or deleting fields from data structures is an open research problem in hot patching [68], and outside the scope of this paper.

Embedded software deceptions. Insider disrupts the attacker's ability to perform reconnaissance by falsifying the information provided to an attacker during network and application fingerprinting. Such information is crucial for the attacker in determining which exploits to deploy against an enterprise network. Insider allows applications to deceive attackers at the endpoint-level. We want to explore this capability further and coordinate it with other layers of the software stack in order to further confuse adversaries.

There is also a need for counter-deception mechanisms that are capable of manipulating advanced attacker deceptions. Heckman et al. [37] discuss a real-time, red team/blue team cyber-wargame experiment that utilized a cyber-deceptive operation in which defenders redirected attackers to a high-interaction honeypot, effectively denying malicious use of the real system while misinforming the adversary with falsified information. Its success should motivate security researchers to examine applications of counter-deception techniques to security defenses. Such techniques should be transparent to users and concealed from adversaries, concurrently limiting attacker gains and increasing the costs of their actions. Promising avenues of research include shielding cyberspace sensors from attackers [27, 70] and exploring inherent asymmetric advantages of using deception in information warfare [45, 71, 83, 89, 90].

Enterprise Deployment. We envision third-party vendors or project maintainers creating JIT patches that can be published to a repository for download by enterprise administrators. The patches can be customized for monitoring, active response, or testing, and managed by automated deployment platforms, such as Ansible Tower [3]. Alerts can be pushed through syslog to a company's Security Information and Event Management (SIEM) system for analysis.

9 CONCLUSION

Insider is a framework for the rapid deployment of JIT security patches into running legacy processes across enterprise networks. By injecting and compiling code inside the running process, Insider quickly installs mini-deceptions that can notify security analysts of attacks, and confuse adversaries with active responses, such as providing misinformation or misdirecting the attacker into an embedded sandbox. We also describe a sandbox architecture, based on Linux namespaces, that can be deployed throughout the network, and transparently envelope attacked processes for monitoring.

We demonstrate the utility of our approach by performing benchmarks on popular enterprise applications. Results suggest that Insider is stable and incurs a small overhead over the stock applications under heavy load. Finally, we present four use cases that show the power and flexibility of the framework. We believe that widespread use Insider's exploit sensing capabilities can significantly reduce the information asymmetry between attacker and defender to level the cyber-battlefield.

REFERENCES

[1] Anomali, Inc. 2014. Modern Honey Network. https://github.com/pwnlandia/mhn. Accessed: 2020-09-02.
[2] Ansible. 2020. https://www.ansible.com/. Accessed: 2020-09-02.
[3] Ansible Tower. 2020. https://www.ansible.com/products/tower. Accessed: 2020-09-02.
[4] Apache HTTP Server. 2019. https://httpd.apache.org/. Accessed: 2019-12-08.
[5] Frederico Araujo, Gbadebo Ayoade, Khaled Al-Naami, Yang Gao, Kevin W. Hamlen, and Latifur Khan. 2019. Improving Intrusion Detectors by Crook-sourcing. In *Proc. Annual Computer Security Applications Conf.* ACM, 245–256.
[6] Frederico Araujo and Kevin W. Hamlen. 2015. Compiler-instrumented, Dynamic Secret-Redaction of Legacy Processes for Attacker Deception. In *Proc. USENIX Security Sym.*
[7] Frederico Araujo, Kevin W. Hamlen, Sebastian Biedermann, and Stefan Katzenbeisser. 2014. From Patches to Honey-Patches: Lightweight Attacker Misdirection, Deception, and Disinformation. In *Proc. ACM Conf. Computer and Communications Security*. 942–953.
[8] Jeff Arnold and M. Frans Kaashoek. 2009. Ksplice: Automatic Rebootless Kernel Updates. In *Proc. European Conf. Computer Systems*. 187–198.
[9] Sina Bahram, Xuxian Jiang, Zhi Wang, Mike Grace, Jinku Li, Deepa Srinivasan, Junghwan Rhee, and Dongyan Xu. 2010. DKSM: Subverting Virtual Machine Introspection for Fun and Profit. In *Proc. IEEE Sym. Reliable Distributed Systems*. 82–91.
[10] Antonio Bianchi, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. 2015. NJAS: Sandboxing Unmodified Applications in Non-rooted Devices Running Stock Android. In *Proc. ACM CCS Work. on Security and Privacy in Smartphones and Mobile Devices*. 27–38.
[11] Bind. 2019. https://www.isc.org/bind/. Accessed: 2019-12-02.
[12] Peter J. Brady, Sergey Bratus, and Sean Smith. 2019. Dynamic Repair of Mission-Critical Applications with Runtime Snap-Ins. In *Proc. Int. Conf. Critical Infrastructure Protection*.
[13] Murray Brand, Craig Valli, and Andrew Woodward. 2010. Malware Forensics: Discovery of the Intent of Deception. *J. Digital Forensics, Security and Law* 5, 4 (2010), 31–42.
[14] Matthew L. Bringer, Christopher A. Chelmecki, and Hiroshi Fujinoki. 2012. A Survey: Recent Advances and Future Trends in Honeypot Research. *Int. J. Computer Network and Information Security* 4, 10 (2012).
[15] Haibo Chen, Jie Yu, Rong Chen, Binyu Zang, and Pen-Chung Yew. 2007. Polus: A powerful live updating system. In *Proc. IEEE Int. Conf. Software Engineering*. 271–281.
[16] Haibo Chen, Jie Yu, Chengqun Hang, Binyu Zang, and Pen-Chung Yew. 2011. Dynamic software updating using a relaxed consistency model. *IEEE Transactions on Software Engineering* 37, 5 (2011), 679–694.
[17] Xu Chen, Jon Andersen, Z. Morley Mao, Michael Bailey, and Jose Nazario. 2008. Towards an Understanding of Anti-virtualization and Anti-debugging Behavior in Modern Malware. In *Proc. IEEE/IFIP Int. Conf. Dependable Systems and Networks*. 177–186.
[18] Yue Chen, Yulong Zhang, Zhi Wang, Liangzhao Xia, Chenfu Bao, and Tao Wei. 2017. Adaptive Android Kernel Live Patching. In *Proc. USENIX Security Sym.* 1253–1270.
[19] Codenomicon. 2014. The Heartbleed Bug. http://heartbleed.com. Accessed: 2020-09-01.
[20] Stephen Crane, Per Larsen, Stefan Brunthaler, and Michael Franz. 2013. Booby Trapping Software. In *Proc. New Security Paradigms Work.* 95–106.
[21] Daniele Cono D'Elia and Camil Demetrescu. 2018. On-Stack Replacement, Distilled. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*. 166–180.
[22] Deutsche Telekom AG. 2015. T-Pot: DTAG Community Honeypot Project. http://dtag-dev-sec.github.io. Accessed: 2020-09-01.
[23] DNSPerf. 2019. https://www.dns-oarc.net/tools/dnsperf. Accessed: 2019-11-28.
[24] Docker. 2020. https://www.docker.com/. Accessed: 2020-09-01.
[25] DoD Comptroller. 2015. *Guidance for Performing Inventory Counts*. Technical Report. U.S. Office of the Under Secretary of Defense, Financial Improvement and Audit Readiness.
[26] Edgescan. 2019. Vulnerability Statistics Report.
[27] Barbara Endicott-Popovsky, Julia Narvaez, Christian Seifert, Deborah A. Frincke, Lori Ross O'Neil, and Chiraag Aval. 2009. Use of Deception to Improve Client Honeypot Detection of Drive-by-download Attacks. In *Proc. Int. Conf. Foundations of Augmented Cognition: Neuroergonomics and Operational Neuroscience*. 138–147.
[28] Fluentd. 2019. http://www.fluentd.org/. Accessed: 2019-12-06.
[29] Charles A. Fowler and Robert F. Nesbit. 1995. Tactical Deception in Air-land Warfare. *J. Electronic Defense* 18, 6 (1995), 37–45.
[30] ftpbench. 2019. https://pypi.python.org/pypi/ftpbench/1.0. Accessed: 2019-12-10.
[31] Cristiano Giuffrida, Anton Kuijsten, and Andrew S Tanenbaum. 2013. Safe and automatic live update for operating systems. In *ACM SIGARCH Computer Architecture News*, Vol. 41. ACM, 279–292.
[32] Glastopf. 2009. Web Application Honeypot. https://github.com/mushorg/glastopf. Accessed: 2020-09-01.
[33] Andy Greenberg. 2018. The untold story of NotPetya, the most devastating cyberattack in history. *Wired* (2018).
[34] Ravit Greister and Daniel Goldberg. 2017. SambaCry, the Seven Year Old Samba Vulnerability, is the Next Big Threat (for now). https://www.guardicore.com/2017/05/samba/. Accessed: 2019-12-12.
[35] Nadav Grossman. 2017. EternalBlue—Everything There Is To Know.
[36] Kristin E. Heckman, Frank J. Stech, Roshan K. Thomas, Ben Schmoker, and Alexander W. Tsow. 2015. *Cyber Denial, Deception and Counter Deception: A Framework for Supporting Active Cyber Defense*. Advances in Information Security, Vol. 64. Springer.
[37] Kristin E. Heckman, Michael J. Walsh, Frank J. Stech, Todd A. O'boyle, Stephen R. DiCato, and Audra F. Herber. 2013. Active Cyber Defense with Denial and Deception: A Cyber-wargame Experiment. *Computers & Security* 37 (2013), 72–77.
[38] Hai Huang, W-K Tsai, and Yinong Chen. 2005. Autonomous hot patching for web-based applications. In *Proc. IEEE Int. Conf. Computer Software and Applications Conference*, Vol. 2. 51–56.
[39] Galen Hunt and Doug Brubacher. 1999. Detours: Binary Interception of Win32 Functions. In *Proc. Conf. on USENIX Windows NT Sym.* USENIX Association.
[40] Suman Jana, Donald E Porter, and Vitaly Shmatikov. 2011. TxBox: Building secure, efficient sandboxes with system transactions. In *Proc. IEEE Sym. Security and Privacy*. 329–344.
[41] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. 2007. Stealthy Malware Detection Through VMM-based Out-of-the-box Semantic View Reconstruction. In *Proc. ACM Conf. Computer and Communications Security*. 128–138.
[42] Kippo. 2009. SSH honeypot. https://github.com/desaster/kippo. Accessed: 2020-09-01.
[43] kpatch. 2020. kpatch: dynamic kernel patching. https://github.com/dynup/kpatch. Accessed: 2020-05-19.
[44] Neal Krawetz. 2004. Anti-honeypot Technology. *IEEE Security & Privacy* 2, 1 (2004), 76–79.
[45] Jamie Lawson, Rajdeep Singh, Michael Hultner, and Kartik B. Ariyur. 2011. Deception Robust Control for Automated Cyber Defense Resource Allocation. In *Proc. IEEE Int. Multi-Disciplinary Conf. Cognitive Methods in Situation Awareness and Decision Support*. 56–59.
[46] Yanlin Li, Jonathan M McCune, and James Newsome. 2014. MiniBox: A Two-Way Sandbox for x86 Native Code. In *Proc. USENIX Annual Technical Conf.*
[47] Linux Programmer's Manual. 2019. capabilities - overview of Linux capabilities. http://man7.org/linux/man-pages/man7/capabilities.7.html. Accessed: 2019-12-13.
[48] Linux Programmer's Manual. 2019. cgroups - Linux control groups. http://man7.org/linux/man-pages/man7/cgroups.7.html. Accessed: 2019-12-12.
[49] Linux Programmer's Manual. 2019. namespaces: Overview of Linux Namespaces. http://man7.org/linux/man-pages/man7/namespaces.7.html. Accessed: 2019-12-12.
[50] LLVM. 2019. http://www.llvm.org/. Accessed: 2019-12-13.
[51] Kristis Makris and Rida A. Bazzi. 2009. Immediate multi-threaded dynamic software updates using stack reconstruction. In *Proc. USENIX Annual Technical Conf.* 397–410.
[52] Bill McCarty. 2003. The Honeynet Arms Race. *IEEE Security & Privacy* 1, 6 (2003), 79–82.
[53] Jonathan M McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. 2010. TrustVisor: Efficient TCB reduction and attestation. In *Proc. IEEE Sym. Security and Privacy*. IEEE, 143–158.
[54] Microsoft IIS. 2019. https://www.iis.net/. Accessed: 2019-11-25.
[55] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. 2006. Practical Dynamic Software Updating for C. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*. 72–83.
[56] NGINX. 2019. https://www.nginx.com/. Accessed: 2019-12-01.
[57] NIST. 2002. Special Publication (SP) 800-40, Procedures for Handling Security Patches. *NIST* (2002).
[58] NIST. 2005. Special Publication (SP) 800-40 Revision 2, Creating a Patch and Vulnerability Management Program. *NIST* (2005).
[59] NIST. 2013. Special Publication (SP) 800-184, Guide for Cybersecurity Event Recovery. *NIST* (2013).
[60] NIST. 2013. Special Publication (SP) 800-40 Revision 3, Guide to Enterprise Patch Management Technologies. *NIST* (2013).
[61] NIST. 2014. The Shellshock Bash Vulnerability. https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-6271. Accessed: 2020-09-02.
[62] NIST. 2020. Common Vulnerability Scoring System. https://nvd.nist.gov/vuln-metrics/cvss. Accessed: 2020-09-05.
[63] Javier Pastor-Galindo, Pantaleone Nespoli, Félix Gómez Mármol, and Gregorio Martínez Pérez. 2020. The not yet exploited goldmine of OSINT: Opportunities, open challenges and future trends. *IEEE Access* 8 (2020), 10282–10304.
[64] Mathias Payer and Thomas R Gross. 2013. Hot-patching a web server: A case study of asap code repair. In *Proc. IEEE Int. Conf. Privacy, Security and Trust.*

[65] Ponemon Institute. 2019. *2019 Cost of Data Breach.* Technical Report. Ponemon Institute.

[66] Niels Provos. 2004. A Virtual Honeypot Framework. In *Proc. USENIX Security Sym.* 1–14.

[67] Niels Provos and Thorsten Holz. 2007. *Virtual Honeypots: From Botnet Tracking to Intrusion Detection.* Addison-Wesley.

[68] Ashwin Ramaswamy, Sergey Bratus, Sean W Smith, and Michael E Locasto. 2010. Katana: A hot patching framework for elf executables. In *Proc. IEEE Conf. Security and Privacy.* 507–512.

[69] Ole André V. Ravnås. 2019. Frida. https://www.frida.re/. Accessed: 2019-12-13.

[70] Mason Rice, Daniel Guernsey, and Sujeet Shenoi. 2011. Using Deception to Shield Cyberspace Sensors. In *Proc. IFIP WG 11.10 Int. Conf. Critical Infrastructure Protection.* 3–18.

[71] Seth Robertson, Scott Alexander, Josephine Micallef, Jonathan Pucci, James Tanis, and Anthony Macera. 2015. CINDAM: Customized Information Networks for Deception and Attack Mitigation. In *Proc. IEEE Int. Conf. Self-Adaptive and Self-Organizing Systems Work.* 114–119.

[72] Florian Rommel, Lennart Glauer, Christian Dietrich, and Daniel Lohmann. 2019. Wait-Free Code Patching of Multi-Threaded Processes. In *Proc. Work. Programming Languages and Operating Systems.*

[73] Neil C. Rowe. 2004. A Model of Deception During Cyber-attacks on Information Systems. In *Proc. IEEE Sym. Multi-Agent Security and Survivability.* 21–30.

[74] Neil C. Rowe. 2006. A Taxonomy of Deception in Cyberspace. In *Proc. Int. Conf. Information Warfare and Security.*

[75] Neil C. Rowe, Binh T. Duong, and E. John Custy. 2006. Fake Honeypots: A Defensive Tactic for Cyberspace. In *Proc. IEEE Information Assurance Work.* 223–230.

[76] Samba. 2019. https://www.samba.org/. Accessed: 2019-12-10.

[77] Sendmail. 2019. http://www.postfix.org/sendmail.1.html. Accessed: 2019-11-29.

[78] Monirul I. Sharif, Andrea Lanzi, Jonathon T. Giffin, and Wenke Lee. 2008. Impeding Malware Analysis Using Conditional Code Obfuscation. In *Proc. Sym. Network and Distributed System Security.*

[79] Steve Souders. 2007. *High Performance Web Sites: Essential Knowledge for Front-End Engineers.* O'Reilly.

[80] Murugiah Souppaya, Kevin Stine, Mark Simos, Sean Sweeney, and Karen Scarfone. 2020. Critical Cybersecurity Hygiene: Patching the Enterprise. *NIST NCCoE* (March 2020).

[81] Lance Spitzner. 2002. *Honeypots: Tracking Hackers.* Addison-Wesley.

[82] Lance Spitzner. 2003. The Honeynet Project: Trapping the Hackers. *IEEE Security & Privacy* 1, 2 (2003), 15–23.

[83] John P. Sullins. 2014. Deception and Virtue in Robotic and Cyber Warfare. In *The Ethics of Information Warfare*, Luciano Floridi and Mariarosaria Taddeo (Eds.). Springer, 187–201.

[84] Sysdig. 2019. http://www.sysdig.org/. Accessed: 2019-12-07.

[85] Olivier Thonnard and Marc Dacier. 2008. A Framework for Attack Patterns' Discovery in Honeynet Data. *Digital Investigation: The Int. J. Digital Forensics & Incident Response (the Proc. Annual Digital Forensics Research Conf.* 5 (2008), S128–S139.

[86] Nikos Virvilis, Bart Vanautgaerden, and Oscar Serrano Serrano. 2014. Changing the Game: The Art of Deceiving Sophisticated Attackers. In *Proc. IEEE Int. Conf. Cyber Conflict.* 87–97.

[87] vsftpd. 2019. https://security.appspot.com/vsftpd.html. Accessed: 2019-12-10.

[88] Ping Wang, Lei Wu, Ryan Cunningham, and Cliff C. Zou. 2010. Honeypot Detection in Advanced Botnet Attacks. *Int. J. Information and Computer Security* 4, 1 (2010), 30–51.

[89] Ben Whitham. 2013. Canary Files: Generating Fake Files to Detect Critical Data Loss From Complex Computer Networks. In *Proc. Int. Conf. Cyber Security, Cyber Peacefare and Digital Forensic.* 170–179.

[90] Ben Whitham. 2014. Design Requirements for Generating Deceptive Content to Protect Document Repositories. In *Proc. Australian Information Warfare Conf.* 20–30.

[91] David Williams-King, Graham Gobieski, Kent Williams-King, James P. Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P. Kemerlis, Junfeng Yang, and William Aiello. 2016. Shuffler: Fast and Deployable Continuous Code Re-randomization. In *Proc. USENIX Sym. Operating Systems Design and Implementation.* 367–382.

[92] Meng Xu, Yeongjin Jang, Xinyu Xing, Taesoo Kim, and Wenke Lee. 2015. UCognito: Private Browsing Without Tears. In *Proc. ACM Conf. Computer and Communications Security.* 438–449.

[93] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009. Native client: A sandbox for portable, untrusted x86 native code. In *Proc. IEEE Sym. Security and Privacy.* 79–93.

[94] Ilsun You and Kangbin Yim. 2010. Malware Obfuscation Techniques: A Brief Survey.. In *Proc. IEEE Int. Conf. Broadband, Wireless Computing, Communication and Applications.* 297–300.