Hot-Patching a Web Server: a Case Study of ASAP Code Repair

Mathias Payer
ETH Zurich and UC Berkeley

Thomas R. Gross
ETH Zurich

***Abstract*—Software updates are the current standard to respond to software bugs. The software developer provides an update fix that is then applied by the administrator: the binary is modified and the service is restarted. Restarting a service inevitably leads to downtime and service unavailability; in the case of a multi-threaded installation of Apache, restart takes several seconds and depending on the load of the web server, several hundred or even thousand client requests will be rejected with an error. Given the cost of restarts, system administrators attempt to minimize the frequency of service restarts or postpone a restart until the next maintenance window.**

However, to ensure the integrity of the system, code repair must happen as soon as possible (ASAP). We describe here the effectiveness of an on-the-fly update system that provides ASAP repair by integrating dynamic patches with a sandbox based on dynamic binary translation. To investigate the feasibility of ASAP code repair, we analyze the software updates released for Apache 2.2 between Dec 1st, 2005 and Feb 18, 2013. The study shows that such a system allows patching 45 of 49 bugs at runtime. Of the 4 unpatchable bugs, 1 bug is not applicable to dynamic update mechanisms, and 3 bugs require a restart. Furthermore, a performance evaluation of the prototype implementation shows that our approach adds low execution overhead (below 7% for different configurations that request a 287kB file).

I. Introduction

Software services are under constant attack so a rapid repair of any vulnerability is crucial to the security of the critical infrastructure. Networked (web) systems are exposed to many forms of attack, so the security and availability of their services demand vigorous installation of any software updates that are made available. However, installing a software update is not free and system administrators face the hard problem of deciding when it is best to update a given service. There is a trade-off between installing a new software patch right away or waiting with the installation until the next maintenance window.

From a security perspective it is necessary to install the patch immediately. In practice, however, system administrators may delay the installation of updates for a variety of reasons: site-specific updates must be performed prior to installation, there may be higher-priority tasks, and/or the installation causes a service to be unavailable for some time, so the installation of the patch is postponed until later. This common "*late update*" practice [7] leaves an open window of opportunity for attackers between the time a bug is discovered and the time the bug is patched in a given running system.

The key to facilitate rapid deployment of security updates is to automate code repair. If the software infrastructure can take care of installing patches, then security updates happen *as soon as possible* – when they become available. In this paper we investigate if this approach is practical, i.e. we analyze the security updates of a significant web service (the Apache server) and investigate if these updates can be handled by an automatic update system.

The rapid installation of software updates builds on two research areas. A *sandbox* protects running applications from security vulnerabilities, and *dynamic software update* mechanisms allow on-the-fly code modification:

A *sandbox* [6], [9], [19], [21] detects attacks and protects the integrity of the system by terminating the running application. Sandboxes build on a form of Software-based Fault Isolation (SFI) [12], [26], [27] and often rely on binary translation [10], [16], [19] to support unmodified binary applications.

On the other hand a *dynamic software update* mechanism [1], [5], [8], [11], [13], [14], [20], [22], [25] installs newly available patches on-the-fly during the execution of the application. Sandboxes have the advantage to detect both known and unknown vulnerabilities but they only protect the integrity of the system and leave out availability. A dynamic software update mechanism on the other hand increases the availability and protects the integrity of an application with the disadvantage that only discovered and patched vulnerabilities are covered while unknown vulnerabilities are still exploitable.

DynSec [17] is a dynamic update mechanism that builds on the TRuE/libdetox [18], [19] software sandbox. The software sandbox ensures that the running application is protected from any known and unknown vulnerabilities while the dynamic update mechanism enables on-the-fly application upgrades without the need to restart the application. Combining a sandbox and a dynamic update mechanism has several advantages: application integrity is protected from known and unknown exploits while the need to restart services during upgrades is removed to increase the availability. In addition, both systems can build on the same binary translation component. The sandbox weaves security checks into the translated application code while the update mechanism checks for security patches during the translation of the application code.

The distribution and specification of the software updates is an orthogonal problem to the application of the update. When package maintainers generate a new security update for a specific software package they already have to generate a patch for their current software version and can generate a binary software patch alongside the updated software package with little overhead. An extension of the package manager that is available in all current Linux distributions can then download these patches and apply them whenever they become available.

This paper investigates the feasibility and practicability of using such a combined sandbox and dynamic update mechanism to provide ASAP code repair – the installation of patches *as soon as possible*. We evaluate all published security patches of the Apache 2.2 webserver to determine if they can be handled by DynSec. Apache is a large and complex multi-threaded and highly parallel software system that supports a large amount of different (software) modules and configurations that are loaded, reloaded, and unloaded at runtime. Apache is deployed to serve more than 340 million websites [15] (54% of all total websites). Of course it is impossible to predict future patches (and if they can be handled by a system like DynSec) but looking at all past security patches over more than 7 years allows us to assess the effectiveness of ASAP code repair. This paper makes the following contributions:

1) Analysis and classification of all 49 security-relevant bugs for Apache 2.2, a large complex real-world software system;
2) A security update case study that evaluates the feasibility to generate patches for a dynamic software update system for all security bugs of Apache versions 2.2.0 to 2.2.24-dev;
3) A performance evaluation of the DynSec prototype for Apache 2.2.

II. A Combined Sandbox and Dynamic Update Mechanism

Sandbox based protection mechanisms and dynamic update mechanisms share one important property: both mechanisms rely on some form of virtualization system to adapt the original application code either to weave additional security checks into the application code or to add the flexibility to replace existing code with new code at specific program locations.

Sandboxes and dynamic software updates are two complementing security mechanisms that protect applications from software vulnerabilities. Sandboxes protect the integrity of the system by extending the application with additional security checks; the application stays vulnerable but exploits are detected and the application is terminated before the system's integrity is compromised. Dynamic security updates on the other hand apply software patches at runtime to remove discovered vulnerabilities, thereby fixing the underlying bug; the vulnerability can no longer be exploited but only the patched vulnerability is fixed. So software patches give stronger guarantees than sandboxing but for a smaller set of vulnerabilities.

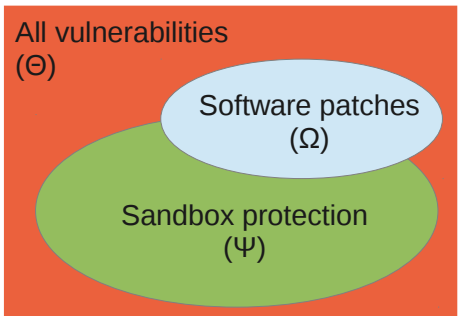

Fig. 1. Comparison between sandbox-based and patching-based security.

Figure 1 shows the sets of all software vulnerabilities Θ of an application, the software vulnerabilities Ψ that are protected by a sandbox, and the software vulnerabilities Ω that are patched compared to a given older version of the application. Both Ψ and Ω are subsets of Θ ($\Psi \subseteq \Theta$ and $\Omega \subseteq \Theta$). In addition, Ψ and Ω overlap partially. While most sandboxes target code integrity, control-flow integrity, and some form of system call policy they cannot protect applications from all forms of vulnerabilities (e.g., full data-flow tracking is generally not feasible due to the large performance overhead, or data-based attacks like SQL injection or XSS vulnerabilities that attack missing sanitizers between applications are out of scope as well). Software updates on the other hand may fix any discovered vulnerability but need specific knowledge of the discovered vulnerability and human effort to generate the new software version (and the update). While the intersection between Ψ and Ω is usually not the empty set (in addition, whenever a sandbox stops a vulnerability the logfile usually indicates the location of the vulnerability in the application code which can be used to generate a software update) there are some vulnerabilities that are only in one of the two sets, e.g., all unknown or unpatched vulnerabilities are in Ψ while many higher-level data-oriented bugs are only covered by Ω.

Related work has studied these two protection mechanisms in isolation. We argue that due to the facts that (i) both mechanisms rely on user-space virtualization and (ii) they protect from different kinds of vulnerabilities with different protection guarantees it makes sense to *combine both protection mechanisms leveraging the same virtualization system*.

In the following four sections we discuss (i) how a sandbox protects benign applications from memory based attack techniques, (ii) dynamic software updating and how the technique falls back to a virtualization system, (iii) our prototype implementation, and (iv) a patch distribution system that can be used to push patches to individual systems.

A. Sandboxing: Software-based Fault Isolation

Sandboxing is a form of Software-based Fault Isolation (SFI) [26] that protects executing code from unkown vulnerabilities. The sandbox isolates untrusted code and ensures that the code cannot damage any structures outside the sandbox. The sandbox uses binary translation to filter instructions, to guard control flow transfers, and to secure memory writes.

Sandboxing enforces the integrity of an application by adding security guards. The virtualization system handles the on-the-fly translation of the executed code. The sandbox collects information from different sources (e.g., from analyzing symbol tables or relocation information which are both available in the binaries and libraries) and instructs the binary translator to weave additional security checks into the translated application code. Dynamic security guards check the target of every control flow transfer (e.g., a shadow stack protects against return oriented programming [23]).

Another advantage of the virtualization system is that all system calls can be redirected to a system call interposition layer. This interposition layer imposes a specific per-application system call policy. If the application tries to execute a system call that is not part of the policy it is terminated.

B. Dynamic Software Updating

In an unprotected system, administrators must balance availability and integrity. Once a bug is discovered and a patch is available, all instances of an application, e.g., all running Apache servers, must be updated to ensure integrity while keeping downtime to a minimum to ensure availability. Most systems, however, are not designed with updateability in mind. Individual modules, e.g., shared libraries, can be reloaded at a *coarse-grained level* at runtime using the functionality provided by the standard loader, but programmers need to reason beforehand which modules can and should be replaced. A significant drawback of the module-based update approach is that synchronization across multiple threads during the update is difficult: the programmer must ensure that no other thread is accessing code (e.g., by executing code directly or indirectly by storing a return address on the stack that points into the code) or state of the module that is reloaded.

Rolling updates, where services in a pool are individually restarted and updated, are an alternative for stateless services. Shared data (e.g., persistent user sessions) between instances becomes a problem due to the possible updated state which must be updated to the new version at one point in time. In addition, the load balancer might become a single point of failure.

DynSec targets (security) updates from a different angle. Instead of transferring both code and state to a new version, the running application's code is replaced with the updated version while keeping the original state, i.e., the data of the application usually remains unchanged. The patch closes the (security) vulnerability and the service runs uninterrupted (i.e., no restart is necessary and the application is only paused during the usually short upgrade process). The administrator updates the service binary and restarts the service at a later point, i.e., during the next scheduled maintenance window.

Leveraging a virtualization system allows DynSec to focus on the updates itself while the virtualization system takes care of (i) resolving addresses, (ii) translating new code, and (iii) hiding the update system from the application using an additional layer of indirection between the original application code and the actual executed code.

When comparing different Apache security updates we made the observation that security updates rarely change (global) data structures. Most observed security updates are local and they fix vulnerabilities by adding runtime checks (e.g., a `NULL` check) or sanitizers (e.g., escaping a parameter string), adjusting existing loop conditions (e.g., fixing an off by 1 error), or replacing unsafe functions with safe counterparts (e.g., `strcpy` is replaced by `strncpy`). We therefore assume that most security updates are local and do not change any global data structures. Due to the semantic gap of using an independent virtualization system DynSec targets local updates only without changes to global data structures or major code refactoring.

Related work usually relies on a-priori changes of the application's source code, changes in the compiler, or a different runtime to make the runtime system aware of the update mechanism. A virtualization system on the other hand can inject the update mechanism into an unaware application (or unaware runtime system).

C. Prototype implementation

DynSec is a software update system that builds on top of a virtualization system. The virtualization system offers the opportunity to translate code as it passes through the dynamic binary translator. Most dynamic binary translators use a code cache to reuse already translated code. A code cache flush forces a retranslation of all existing code, in addition, the translator can then patch the application code (and all libraries) on-the-fly while the code is being translated. The update mechanism handles the distribution and the coordination of the patches and the binary translator handles the patch application. With two small changes in the binary translation system (coordinating the code flush and checking for every translated instruction if a patch exists) it is possible to patch running applications dynamically. DynSec design decisions and a first prototype are discussed in [17].

The prototype implementation of DynSec builds on TRuE/libdetox [18], [19], a sandbox and user-space virtualization system for the execution of potentially unsafe programs that extracts information about the application (and its libraries) by using a trusted loader component that replaces the standard loader. All application code is protected by a dynamic binary translator that weaves additional security guards into the executed code. DynSec extends TRuE by (i) adding several hooks in the binary code translator, (ii) modifying the translator to add (virtual) safe-points to coordinate all application threads for incoming security updates, and (iii) adding a coordination thread that loads, verifies, and coordinates incoming security updates.

The patching thread of DynSec is started as part of the trusted computing base and waits for incoming patches by watching a specific directory (using the `inotify` system call. The trusted computing base contains the trusted loader, the sandbox, DynSec, and the binary translator. Incoming patches are decoded and added to the patch information data structure that is shared among all threads (each application thread uses its own binary translator and code cache). The patching thread synchronizes all application thread and signals them to flush their code caches. The translator thereby indirectly applies the patches by retranslating all code. Instruction patching happens before security hardening; patched instructions and additional functionality in the patch library are security hardened along with the regular application code.

For the current prototype implementation the patches are encoded in a simple binary format. Each patch starts with the number of patch locations; each patch location consists of (i) the address that is patched relative to the executable or shared library (loader information is used during patch application to calculate the absolute address), (ii) the length of the patched location, (iii) the address of the translated instruction in the new version of the application or shared library (to resolve relative references using the dynamic loader), (iv) the length of the new instruction(s), and (v) the machine code representation of the new instruction(s). Patches may also specify a shared library to include additional functionality. The library is loaded into a patch-local context and the symbols in that library are only visible to the patched instructions.

D. Patch generation and distribution

Most Linux distributions already use an automatic update mechanism that chronically checks the main package repository for updates for all installed packages. The system administrator is notified and must agree to install the updated packages. During the update process the services are restarted automatically. This human-in-the-loop process leads to a time lag between the time an update becomes available and the time the update is installed. We argue that an automatic update system can automatically apply these updates on-the-fly without manual interaction or the need to restart the service. An administrator can then update the binary of the software during the next maintenance window.

When a bug for a software package is discovered it is first fixed by the software maintainers (e.g., the Apache maintainers). The package maintainers of individual distributions (e.g., Debian maintainers) pick up the new source code or the patch and generate a new package that is shipped to the customers.

To allow stable functionality and configuration most distributions freeze the software version for a specific release of a distribution. Package maintainers back port security patches to these frozen software versions. For a specific release of the distribution the runtime system (e.g., the version of the compiler, the libc, and other libraries) is fixed and all software is compiled with the same compilation settings. The stable runtime system results in low variance between the patched and the unpatched version and contains mostly the patched region. There are some additional changes due to compiler optimizations (e.g., different inlining settings due to the length of a patched function). This setting is a perfect opportunity to automatically extract binary patches that allow hot patching.

The package maintainers already have the source patch between the old version of the software and the new version and they have the old binary as well. With little additional effort the package maintainers can generate binary patches that fix the software vulnerability. As part of the build system the package maintainers can ensure the functionality of the extracted binary patches before releasing the patches.

The (cryptographically signed) binary patches are then distributed using a similar distribution system like for the upgraded software packages where individual systems can chronologically check for newly available patches and apply them ASAP.

III. Evaluation

Three aspects are important to test the feasibility of a dynamic update mechanism: (i) completeness: does the system support the targeted security patches, (ii) correctness: the dynamic update system should not introduce any bugs or data races, and (iii) performance: the performance overhead for the complete system should be low.

The completeness and feasibility of DynSec is evaluated by providing a detailed study off all security relevant bugs of Apache 2.2 over the span of more than 7 years.

When working on the binary-only level it is hard to reason about correctness and it is out of the scope of this paper to proof the correctness of the dynamic update mechanism. The prototype implementation is successfully tested using a large set of patches that replace different parts of the running application, testing both the replacement of code that is currently executing or on the stack and code that is not currently executed.

Performance overhead is an important metric that greatly influences if a system is used in practice. The performance overhead for DynSec is both evaluated using an Apache-based benchmark and using the SPEC CPU2006 benchmarks. All benchmarks are executed on an Intel Core 2 Quad Q6600 with 2.64GHz and 8GB RAM on Ubuntu 11.04 with Linux kernel 2.6.38 and the GNU compiler collection 4.5.1.

A. Apache 2.2 security study

Apache 2.2 is a mature web-server for medium to large scale web sites. Between the first release of version 2.2.0 on December 1st 2005 and the current release 2.2.24-dev on February 18th 2013, 49 security related bugs are reported[1] and fixed in later releases. The bugs are classified according to their security impact, possible impact levels for bugs are[2] *low*, *moderate*, *important*, and *critical*. This study looks at all security-related bugs for Apache 2.2.0 until the (current) 2.2.24-dev release. The study analyzes the security patches and evaluates if DynSec supports the dynamic application of these patches.

The study separates patches based on complexity and functionality needed in the dynamic updating system. Patches are separated into the following classes:

Simple patch: only few (between 1 and 20) individual instructions change. The patch does not touch any symbols (functions or variables). These patches can be applied without additional shared objects.

New import patch: the patch changes several instructions and imports one or more new functions in the object. The original patch is implemented as one or more functions in a (small) shared library. The shared library is loaded by the patching thread using the trusted loader (that handles the imports as well). The original code locations are then patched with a control flow transfer to the new functions using patched instructions.

New function patch: the patch adds a new function. This new function is implemented in a shared library (similar to the *new import patch*) and the original code locations are patched with a control flow transfer to the new functions.

Additional call patch: the patch adds one or more calls to functions that are already available in the unpatched object. To simplify the implementation each call is implemented as a function in a shared library (similar to the *new import patch*).

New string patch: the patch uses an additional static string that is added to the runtime image through a shared library (similar to the *new import patch*).

Other: the patch cannot be implemented easily for DynSec (e.g., because of type changes or additional initialization code). These patches are problematic for DynSec due to the structural changes in the application.

[1]The Apache homepage lists all the security bugs of the 2.2 release at http://httpd.apache.org/security/vulnerabilities_22.html.

[2]The Apache homepage http://httpd.apache.org/security/impact_levels.html gives a description of the different impact levels.

Fixed in	CVE number	Description	Impact	Type	Patch class	DynSec	Sandbox
2.2.2	CVE-2005-3352	mod_imap referer XSS	moderate	XSS	new import patch	yes	no
2.2.2	CVE-2005-3357	mod_ssl access control	low	DoS	simple patch	yes	no
2.2.3	CVE-2006-3747	mod_rewrite off-by-one	important	EXE	simple patch	yes	yes
2.2.6	CVE-2007-1863	mod_cache NULL deref.	moderate	DoS	simple patch	yes	yes
2.2.6	CVE-2007-1862	mod_cache info. leak	moderate	IL	new function patch	yes	no
2.2.6	CVE-2007-3304	signals to arbitrary pids	moderate	lDoS	add. types, code refactoring	no	yes
2.2.6	CVE-2006-5752	mod_status XSS	moderate	XSS	new string patch	yes	no
2.2.6	CVE-2007-3847	mod_proxy crash	moderate	DoS	new import patch	yes	yes
2.2.8	CVE-2007-5000	mod_imagemap XSS	moderate	XSS	additional call patch	yes	no
2.2.8	CVE-2007-6388	mod_status XSS	moderate	XSS	new import patch	yes	no
2.2.8	CVE-2007-6421	mod_proxy_balancer XSS	low	XSS	new import patch	yes	no
2.2.8	CVE-2007-6422	mod_proxy_balancer DoS	low	DoS	simple patch	yes	yes
2.2.8	CVE-2008-0005	mod_proxy_ftp UTF-7 XSS	low	XSS	add. type, new functions	no	no
2.2.9	CVE-2008-2364	mod_proxy_http inf. loop	moderate	DoS	new imports patch	yes	no
2.2.9	CVE-2007-6420	mod_proxy_balancer CSRF	low	CSRF	new imports patch	yes	no
2.2.10	CVE-2008-2939	mod_proxy_ftp XSS	low	XSS	additional call patch	yes	no
2.2.10	CVE-2010-2791	mod_proxy_http timeout IL	important	IL	additional call patch	yes	no
2.2.12	CVE-2009-0023	APR-util heap underwrite	moderate	HBUF	simple patch	yes	yes
2.2.12	CVE-2009-1955	APR-util XML DoS	moderate	DoS	new import patch	yes	no
2.2.12	CVE-2009-1956	APR-util off-by-one ovfl.	moderate	DoS	simple patch	yes	no
2.2.12	CVE-2009-1195	AllowOverride bypass	low	lPE	simple patch	yes	no
2.2.12	CVE-2009-1891	mod_defalte DoS	low	DoS	new import patch	yes	no
2.2.12	CVE-2009-1191	mod_proxy_ajp info. leak	important	IL	simple patch	yes	no
2.2.12	CVE-2009-1890	mod_proxy rev. proxy DoS	important	DoS	new import patch	yes	no
2.2.13	CVE-2009-2412	APR apr_palloc heap ovfl.	low	HBOF	simple patch	yes	yes
2.2.14	CVE-2009-2699	Solaris pollset DoS	moderate	DoS	simple patch	yes	no
2.2.14	CVE-2009-3095	mod_proxy_ftp cmd. inject.	low	ACI	additional call patch	yes	no
2.2.14	CVE-2009-3094	mod_proxy_ftp NULL deref.	low	DoS	new function patch	yes	yes
2.2.15	CVE-2010-0408	mod_proxy_ajp DoS	low	DoS	simple patch	yes	no
2.2.15	CVE-2010-0434	mode_headers DoS	low	DoS/IL	additional call patch	yes	no
2.2.15	CVE-2010-0425	mod_isapi unload flaw	important	EXE	simple patch	yes	yes
2.2.16	CVE-2010-1452	mod_cache & mod_dav DoS	low	DoS	simple patch	yes	no
2.2.16	CVE-2010-2068	mod_proxy_http timeout	important	IL	simple patch	yes	no
2.2.17	CVE-2010-1623	apr_brigade_split_line DoS	low	DoS	new function patch	yes	no
2.2.17	CVE-2009-3560	expat DoS	low	DoS	simple patch	yes	no
2.2.17	CVE-2009-3720	expat DoS	low	DoS	simple patch	yes	no
2.2.19	CVE-2011-0419	apr_fnmatch flaw	moderate	DoS	new functions patch	yes	no
2.2.20	CVE-2011-3192	range header DoS	important	DoS	new functions patch	yes	no
2.2.21	CVE-2011-3348	mod_proxy_ajp DoS	moderate	DoS	simple patch	yes	no
2.2.22	CVE-2011-3368	mod_proxy rev. proxy	moderate	AIH	new functions	yes	no
2.2.22	CVE-2012-0053	cookie exposure	moderate	IL	new function patch	yes	no
2.2.22	CVE-2011-4317	mod_proxy rev. proxy	moderate	AIH	simple patch	yes	no
2.2.22	CVE-2012-0031	scoreboard parent DoS	low	DoS	type change, new functions	no	no
2.2.22	CVE-2012-0021	mod_log_config NULL deref.	low	DoS	simple patch	yes	no
2.2.22	CVE-2011-3607	mod_setenvif priv. escal.	low	IOF	simple patch	yes	yes
2.2.23	CVE-2012-0883	Insecure var. in startup script	low	lPE	not applicable	NA	no
2.2.23	CVE-2012-2687	mod_negotiation XSS	low	XSS	new import patch	yes	no
2.2.24-dev	CVE-2012-4588	mod_proxy_balancer XSS	moderate	XSS	new import patch	yes	no
2.2.24-dev	CVE-2012-3499	unescaped hostnames XSS	low	XSS	new import patch	yes	no

TABLE I. LIST OF ALL APACHE 2.2 SECURITY BUGS FROM VERSION 2.2.0 UNTIL THE CURRENT VERSION 2.2.24-DEV. THE TABLE USES THE FOLLOWING ABBREVIATIONS: DOS DENIAL OF SERVICE; EXE ARBITRARY CODE EXECUTION; IL INFORMATION LEAK; LDOS LOCAL DENIAL OF SERVICE; XSS CROSS SITE SCRIPTING; CSRF CROSS SITE REQUEST FORGERY; HBUF HEAP BUFFER UNDERFLOW; LPE LOCAL PRIVILEGE ESCALATION; HBOF HEAP BUFFER OVERFLOW; ACI ARBITRARY COMMAND INJECTION; AIH ACCESS TO INTERNAL HOSTS; IOF INTEGER OVERFLOW.

Table I shows a list of all security relevant bugs with the Apache release where the bug is fixed, CVE number of the bug, a short description, impact level, bug type, how the bug is fixed, if the patch can be applied in DynSec, and if a sandbox protects from the bug.

Four patches out of 49 cannot be applied (CVE-2007-3304, CVE-2008-0005, CVE-2012-0031, and CVE-2012-0883). The first three patches require new types, changed types, or code refactoring. Such patches cannot be generated easily. A programmer could write special initialization functions for such patches to implement a specific patch that fixes the bug without additional types. A drawback of this approach is that the DynSec patch construction would consume additional human resources that are better spent on the security patch itself. The last patch, CVE-2012-0883, is not applicable to a dynamic updating system because the patch only changes a startup script.

Out of 49 patches there are 45 patches that can be applied to a running Apache instance using the DynSec framework. 20 patches are simple patches that change non-control flow instructions only, 25 patches need an additional shared library for added functionality, new functions, imported functions from other modules, or calls to existing functions in the same module.

The sandbox stops all discovered control flow integrity attacks: 10 out of 49 discovered vulnerabilities are protected by the sandbox (by safely terminating the Apache process if an attack is detected) until a patch is applied. The protection of the sandbox is restricted to system call policies, code integrity, and control flow integrity checks and is unable to stop information leaks or cross-site-scripting attacks.

Examining the individual patches shows that a large majority of the security related patches are compatible to the DynSec framework. The DynSec patches can be constructed with minimal programmer overhead.[3] During the analysis of the patches we observed that most security updates do not change any data and a low-level virtualization system is suited to replace the application code of binary-only applications.

B. DynSec performance

The performance of the DynSec prototype implementation for server applications is evaluated using the `ab` Apache benchmark and Apache version 2.2.21. The Apache webserver uses multiple processes and multiple threads to serve http requests in parallel. The Apache webserver supports modules as extensions to implement additional functionality (e.g., PHP scripting, WebDav access, or database access).

This benchmark uses 5 different runtime configurations to evaluate the overhead of the sandbox, the overhead of DynSec, and the relative patching overhead. The five configurations are: (i) *native*, a native configuration without sandboxing, (ii) *sandbox*, a configuration with active sandboxing, (iii) *DS-0*, a configuration with an active sandbox and DynSec without patched instructions, (iv) *DS-5k*, a configuration with 5,000 patched instructions in the Apache core, and (v) *DS-10k*, a configuration with 10,000 patched instructions in the Apache core. The files on the Apache webserver were accessed using the `ab` benchmarking tool that is part of the Apache suite. `ab` is executed with 4 concurrent connections. The benchmark uses two different files to evaluate transmission speed of the webserver, `index.html`, a 44 bytes text file, and `picture.png`, a 287kB binary file. Transmission speed is evaluated for 100, 1,000, and 10,000 requests. For each thread the first request after patch application (and the resulting flush of the code cache) experiences additional latency due to the retranslation of the executed code. The additional latency for retranslation is in the noise and not distinguishable from regular latency of the service.

Figure 2(a) and Figure Figure 2(b) show the different Apache configurations. Both figures show that the overhead for DynSec is comparable to the sandboxing overhead. Comparing the different configurations along the X-axis shows that the three patching configurations have roughly the same overhead as the sandbox configuration. The slowdown in throughput for all configurations of DynSec is below 7% for `picture.png`. For `index.html` the throughput is decreased by 15% for 10.000 requests and by 27% for 1.000 requests.

The benchmark run with 100 requests for `index.html` in Figure Figure 2(a) shows the only outlier where the overhead for binary translation is significant compared to the native execution (the throughput is decreased by 59% for DS-0 and 72% for DS-10k). Due to the very low number of requests and the small file (i.e., the CPU spends only little time on I/O in the kernel) the translation and patching overhead cannot be amortized during the short execution time. Larger files and/or more requests reduce the relative overhead for the first translation.

Comparing the different number of requests shows that the overhead for sandboxing and DynSec is amortized through multiple executions of the already translated code. The number of requests is inverse-proportional to the overall overhead of the patching configuration. In server applications the I/O overhead and the time spent in translated code dominates the translation overhead. The low overhead for a high number of requests shows that DynSec is a valid approach to protect long running server applications.

In an additional experiment we evaluate the performance overhead for SPEC CPU2006 benchmarks. On average the TRuE runtime environment results in a slowdown of 11% compared to native performance. DynSec reports 11% slowdown as well and therefore adds no measurable overhead for long running CPU-bound applications. The usual culprits 400.perlbench, 458.sjeng, 464.h264ref, and 453.povray are the benchmarks with most overhead due to the high amount of indirect control flow transfers. Most of the overhead can be attributed to the binary translation system. Both the additional security checks and the dynamic update mechanism only add little overhead on top of the dynamic binary translation.

IV. Related work

In this section we discuss related work in sandboxing/binary translation and dynamic update systems. Sandboxing protects the application at runtime from unkown vulnerabilities while dynamic update systems patch discovered vulnerabilities.

[3] An experienced programmer with knowledge of the DynSec system spends between 5 and 20 minutes to generate a DynSec patch based on the CVE information and the binary diff between the two binaries.

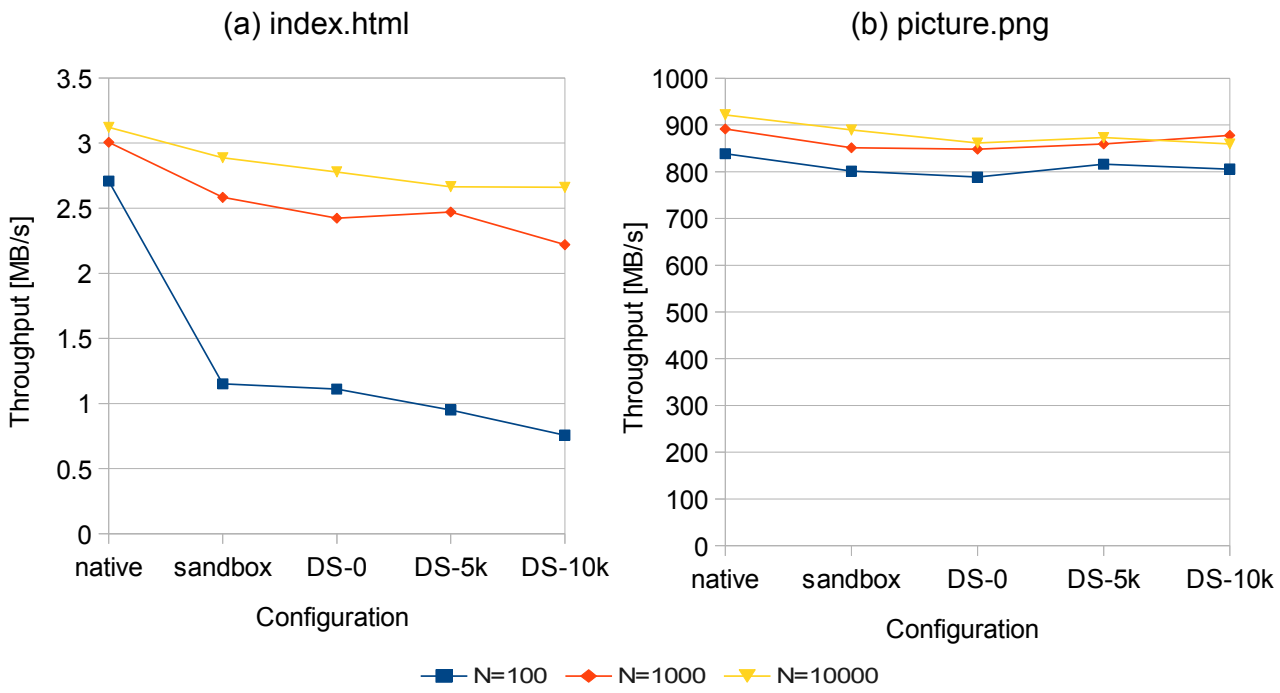

Fig. 2. On the left figure (a) shows the throughput for different configurations of DynSec for `index.html` and on the right figure (b) shows the throughput for different configurations of DynSec for `picture.png`

Dynamic sandboxing approaches like Vx32 [6], Program shepherding [9], Strata [21], and libdetox/TRuE [18], [19] use dynamic binary translation to sandbox the application on-the-fly. Code is translated right before it is executed, usually on a basic block level. Translated basic blocks are placed in a code cache. Security guards are added during the translation and executed from this code cache.

Dynamic software updating [22] relies on some form of a runtime system to load and inject new patches into a running application (or operating system). Compiler-based approaches [1], [2], [5], [8], [11], [13], [14], [25] modify the compiler tool-chain to include the update mechanism into the compiled program while other approaches modify the programming language [8]. Smith et al. [24] give a great overview of different dynamic software updates.

All kernel and user-space based dynamic update systems require a runtime system that is aware of the source-code (of the application and of all used libraries) and prepares the executable for incoming changes already at the compilation stage, thereby limiting the updateability for libraries that were compiled with a different compiler or in a different setting. DynSec, on the other hand, does not need any source-code information and uses user-space virtualization to inject a runtime system into the unmodified, binary-only application. The runtime system must not be prepared and pre-linked into the binary but can be added at any time.

The different runtime systems run at very different levels of granularity. The simplest runtime systems replace the complete runtime image [5], [8] while other systems work on function level [1], [4], or on the basic block level [11]. DynSec works on the most fine-grained level of individual instructions and the evaluation of Apache security patches shows that a majority of the patches can be applied by replacing only a few instructions.

Efficiently synchronizing patch application among multiple threads is a hard problem as some threads may be sleeping in the kernel (e.g., for high latency I/O operations, or because they are waiting for a lock or semaphore) and the dynamic software update mechanism must ensure atomic patch application for correctness but should reduce the delay until the patch is applied and minimize the time needed to apply the patch. Many existing systems therefore do not support multi-threaded applications and state that they support only single-threaded applications [8], [14], [22]. Using a sandbox-based virtualization system that already supports per-thread code caches and system call interposition naturally allows multi-thread patch application by (i) interposing the return of all system calls for threads that are currently sleeping in the kernel and (ii) using a safe-point mechanism at the end of each basic block to synchronize all running threads with negligible overhead.

A virtualization system enables a high-level view of the application and offers an additional level of abstraction between the application and the runtime system. This abstraction enables a clear separation between the patch and the patched application. Another advantage of the virtualization approach is that the application can be updated even if the replaced function is currently being executed (or is on the stack). DynSec injects new code on instruction level granularity and not on function level granularity like other approaches.

LUCOS [4] and Ksplice [2] implement dynamic security updates for Linux kernels. Ksplice modifies the compilation

toolchain to implement a dynamic security update system. LUCOS is based on Xen [3] and uses full-system virtualization to apply kernel updates. LUCOS [4] is the only related dynamic software updating mechanism using virtualization. LUCOS works on function level granularity and modifies page-tables to overwrite function prologues with jump instructions to patched functions. DynSec on the other hand uses user-space virtualization to dynamically patch services on instruction level granularity.

Dynamic updating systems need to address the trade-off between flexibility (i.e., enabling as many updates as possible) versus safety (i.e., ensuring the correctness of the patches and the safety of the application after the patching process). DynSec offers maximum flexibility by allowing a programmer to change any instruction in a patch. The programmer must still ensure that the patch is correct and that it will not crash the system (similar to OPUS). Future work includes the development of an automatic patch extraction process only based on the binary itself whilst ensuring correctness.

V. Conclusion

This paper shows that using a dynamic update mechanism for long running network services like the Apache web server is feasible and highly effective: out of 49 security critical bugs for Apache 2.2, released over more than 7 years, 45 can be patched at runtime. Out of the 4 unpatchable bugs, 1 bug changes only a startup script, and 3 patches require a service restart due to changed data-structures or code refactoring. Of course, the findings may differ for other applications or architectures. However, the results reported for a real-life large-scale software system that is used by many people increase our confidence that the approach is highly effective and that "ASAP code repair" is practical.

Implementing the dynamic update mechanism on top of a virtualization system like a dynamic binary translation system has several advantages: (i) the patching infrastructure can be implemented on top of the virtualization system, reusing the existing virtualization infrastructure and simplifying the dynamic update mechanism; (ii) the virtualization system can be used to protect the integrity of the application by executing additional security checks alongside the patching infrastructure. In addition, the evaluation shows that important security bugs cannot be protected by a sandbox alone and a combination of sandbox and dynamic software update mechanism is necessary to protect a running software service from different classes of attack.

Dynamic updating for unmodified binary applications can be used in real-life settings, and as long as the distribution of patches is part of the strategy to deal with vulnerabilities that are identified after applications have been shipped, a dynamic updating service combined with a software sandbox provides an effective and attractive approach to increase software availability and integrity.

Acknowledgments

We thank the anonymous reviewers for their valuable comments and suggestions and Boris Bluntschli for his help on the implementation of the first prototype of DynSec. Supported, in part, through DARPA award HR0011-12-2-005 (Mathias Payer's work at the University of California at Berkeley).

References

[1] Altekar, G., Bagrak, I., Burstein, P., and Schultz, A. Opus: Online patches and updates for security. In *SSYM'05: In 14th USENIX Security Symp.*

[2] Arnold, J., and Kaashoek, M. F. Ksplice: automatic rebootless kernel updates. In *EuroSys'09*.

[3] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. In *SOSP'03*.

[4] Chen, H., Chen, R., Zhang, F., Zang, B., and Yew, P.-C. Live updating operating systems using virtualization. In *VEE'06*.

[5] Chen, H., Yu, J., Chen, R., Zang, B., and chung Yew, P. Polus: A powerful live updating system. In *ICSE'07*.

[6] Ford, B., and Cox, R. Vx32: lightweight user-level sandboxing on the x86. In *USENIX ATC'08*.

[7] Frei, S., Schatzmann, D., Plattner, B., and Trammell, B. Modeling the security ecosystem-the dynamics of (in) security. *Economics Info. Sec. Privacy* (2010), 79–106.

[8] Hicks, M., Moore, J. T., and Nettles, S. Dynamic software updating. In *PLDI'01*.

[9] Kiriansky, V., Bruening, D., and Amarasinghe, S. P. Secure execution via program shepherding. In *SSYM'02*.

[10] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI'05*.

[11] Makris, K., and Bazzi, R. A. Immediate multi-threaded dynamic software updates using stack reconstruction. In *Usenix ATC'09*.

[12] McCamant, S., and Morrisett, G. Evaluating SFI for a CISC architecture. In *SSYM'06*.

[13] Neamtiu, I., and Hicks, M. Safe and timely updates to multi-threaded programs. In *PLDI'09*.

[14] Neamtiu, I., Hicks, M., Stoyle, G., and Oriol, M. Practical dynamic software updating for C. In *PLDI'06*.

[15] Netcraft. March 2013 webserver study. http://news.netcraft.com/archives/2013/03/01/march-2013-web-server-survey.html, 2013.

[16] Nethercote, N., and Seward, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI'07*.

[17] Payer, M., Bluntschli, B., and Gross, T. R. DynSec: On-the-fly code rewriting and repair. In *HotSWUp'13*.

[18] Payer, M., and Gross, T. R. Fine-grained user-space security through virtualization. In *VEE'11*.

[19] Payer, M., Hartmann, T., and Gross, T. R. Safe loading - a foundation for secure execution of untrusted programs. In *S&P'12: Proc. Int'l Symp. on Security and Privacy* (2012).

[20] Popovici, A., Gross, T., and Alonso, G. Dynamic weaving for aspect-oriented programming. In *AOSD'02*.

[21] Scott, K., and Davidson, J. Safe virtual execution using software dynamic translation. *ACSAC'02*.

[22] Segal, M. E., and Frieder, O. Dynamic program updating: a software maintenance technique for minimizing software downtime. *Journal of Software Maintenance'89*.

[23] Shacham, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *CCS'07: Proc. 14th Conf. on Computer and Communications Security* (2007), pp. 552–561.

[24] Smith, E. K., Hicks, M., and Foster, J. S. Towards standardized benchmarks for dynamic software updating systems. In *HotSWUp'12*.

[25] Stoyle, G., Hicks, M., Bierman, G., Sewell, P., and Neamtiu, I. Mutatis mutandis: Safe and predictable dynamic software updating. *TOPLAS'07*.

[26] Wahbe, R., Lucco, S., Anderson, T. E., and Graham, S. L. Efficient software-based fault isolation. In *SOSP'93*.

[27] Yee, B., Sehr, D., Dardyk, G., Chen, J. B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., and Fullagar, N. Native client: A sandbox for portable, untrusted x86 native code. In *IEEE S&P'09*.