Retroactive Auditing

Xi Wang Nickolai Zeldovich M. Frans Kaashoek
MIT CSAIL

ABSTRACT

Retroactive auditing is a new approach for detecting past intrusions and vulnerability exploits based on security patches. It works by spawning two copies of the code that was patched, one with and one without the patch, and running both of them on the same inputs observed during the system's original execution. If the resulting outputs differ, an alarm is raised, since the input may have triggered the patched vulnerability. Unlike prior tools, retroactive auditing does not require developers to write predicates for each vulnerability.

1. INTRODUCTION

Due to the increasing size and complexity of software, software defects have become inevitable [2]. These defects often lead to vulnerabilities, and allow an adversary to break into the system, until developers release updates that fix the defects and administrators patch the system with those updates. Once the administrators install an update, they may want to know whether some adversary already exploited the corresponding vulnerability before the patch was installed.

This position paper proposes a new tool, RAD, to help administrators audit the past execution of their system and detect intrusions. RAD's workflow is shown in Figure 1. After a patch is released, RAD re-executes all programs that invoked the vulnerable code, both with and without the patch applied, and feeds in the inputs seen during the system's original execution. If the programs behave differently, an adversary may have exploited this vulnerability. As we describe in Section 6, unlike previous approaches [5], RAD does not require developers to write predicates for each vulnerability.

The key challenge facing RAD is reducing false alarms. Executing the patched code may lead to a different bit-level system state, even if that state is functionally identical to the original one. For example, the program may be non-deterministic, in which case each run would produce a different result. A program might also create a complex data structure, such as a binary tree, which may end up having different pointer values, due to differences in `malloc`'s behavior, even though the trees are equivalent.

The main contribution of this paper is the idea of retroactive auditing. Based on a visual inspection of past vulnerabilities discovered in the Apache web server, we argue that this approach should work well in practice. We further propose several ideas for reducing false alarms.

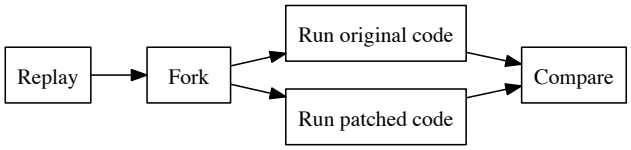

Figure 1: The workflow of retroactive auditing.

The rest of the paper is organized as follows. Section 2 gives an overview of RAD. Section 3 presents a feasibility study using vulnerabilities from Apache, based on visual inspection. Section 4 shows some initial results for applying RAD to two vulnerabilities. Section 5 discusses future challenges. Section 6 relates RAD to previous work and Section 7 concludes.

2. OVERVIEW

This section describes how RAD works, using CVE-2009-0023 shown in Figure 2 as a running example. This vulnerability was discovered in the APR-util library bundled with the Apache web server. At line 7, the developers mistakenly used a signed integer `(int)s[i]` as an index of array `shift`; the index could be negative, which allows an adversary to craft an input string to overwrite heap memory. Line 14 contains a similar bug. Patching the problem is straightforward: use an unsigned index. RAD's goal is to apply the patch retroactively to a system and determine if an attacker used the vulnerability to compromise the system.

2.1 A strawman approach

Consider a simple command-line text search program `xgrep`, which reads from stdin and prints results to stdout, just like standard `grep`, except that it makes use of the vulnerable function in our running example (see Figure 2). We assume that the administrator has deployed a monitoring infrastructure that records the past execution of the system and is able to replay the system's execution from a previous state (e.g., using Retro [7]). To determine whether any past execution of `xgrep` was exploited, a strawman version of RAD works as follows.

First, RAD rolls the system back to a state before the exploit could have occurred (e.g., when the vulnerable software was first installed).[1] Then, RAD replays the system's execution using the inputs recorded during the original run. During replay, before each execution of `xgrep`, RAD spawns both a copy of `xgrep` using the

[1]Administrators may need to make trade-offs between a longer time horizon for auditing and higher storage costs for audit logs.

```
--- apr/apr-util/branches/1.3.x/strmatch/apr_strmatch.c ...
+++ apr/apr-util/branches/1.3.x/strmatch/apr_strmatch.c ...
@@ -103,12 +103,13 @@
 if (case_sensitive) {
   pattern->compare = match_boyer_moore_horspool;
   for (i = 0; i < pattern->length - 1; i++) {
-    shift[(int)s[i]] = pattern->length - i - 1;
+    shift[(unsigned char)s[i]] = pattern->length - i - 1;
   }
 }
 else {
   pattern->compare = match_boyer_moore_horspool_nocase;
   for (i = 0; i < pattern->length - 1; i++) {
-    shift[apr_tolower(s[i])] = pattern->length - i - 1;
+    shift[(unsigned char)apr_tolower(s[i])]
+      = pattern->length - i - 1;
   }
 }
```

Figure 2: The patch for CVE-2009-0023, heap underwrites in APR-util function `apr_strmatch_precompile`.

original, vulnerable APR-util library, and a copy of `xgrep` using the patched APR-util library. RAD then runs the two processes, collects the outputs from both runs, and compares their outputs. If there is no difference, RAD concludes that the vulnerable function was not exploited; otherwise RAD reports a possible exploit. This conclusion assumes that the patch was designed to address the vulnerability and that the patch addresses the vulnerability correctly.

The strawman approach re-executes any process, in its entirety, that may have been affected by the patched code. This poses several challenges when applying the same approach to a complex program like the Apache web server. First, it is costly to re-execute the entire process, and it is often unnecessary, because a patch is often small [10] and typically causes small changes in a program's behavior. Second, programs such as Apache often involve several threads or processes, and their behavior can be non-deterministic even with the same inputs. As a result, it is difficult to match two non-deterministic runs, even if the patch does not functionally change anything.

2.2 Fine-grained auditing

RAD addresses the above challenges by using fine-grained re-execution and auditing. Instead of auditing the whole process, RAD can audit runs of a smaller code piece that is likely to be deterministic (e.g., a single function in Figure 2) as follows.

Again, RAD rolls back the system to a correct state, and replays the past executions of vulnerable processes. Since the patch modifies only one function, `apr_strmatch_precompile`, RAD intercepts every call to that function in all processes (currently, it uses `LD_PRELOAD` on Linux to inject a code stub).

Before every invocation of the vulnerable function, the injected code stub forks the calling process into two: one fork invokes the vulnerable function and the other fork invokes the patched function. RAD runs the two versions of the function and records the memory writes they perform during their execution. This recording is implemented using Pin [8], which can instrument load and store instructions.

RAD compares the resulting system state of the two processes, by doing a diff of the recorded memory writes. In our example, if an adversary did not compromise the vulnerable function, then the stores to the array `shift` are within its boundary. Thus both versions of the function will behave in the same way (e.g., there will be no difference in the recorded writes), and RAD will report no warnings. If an adversary enticed the program to use a negative index, however, the underwrites of `shift` will happen in the vulnerable code, but not in the patched code. In this case the diff between the recorded

```
--- modules/proxy/mod_proxy_ftp.c ...
+++ modules/proxy/mod_proxy_ftp.c ...
@@ -385,4 +385,5 @@
 if (wildcard != NULL) {
+  wildcard = ap_escape_html(p, wildcard);
   APR_BRIGADE_INSERT_TAIL(out, apr_bucket_pool_create(
     wildcard, strlen(wildcard), p,
     c->bucket_alloc));
```

Figure 4: The patch for CVE-2008-2939, cross-site scripting.

memory writes will be non-empty, and RAD will report an exploit and output the diff (see Section 4.1).

This fine-grained auditing scheme imposes additional requirements on patches. Particularly, a patch should not change the function signature, such as adding or removing parameters, nor should it change the layout of any external data structures used by the function. Otherwise, the patched code cannot run starting with the same inputs from the state after fork. For patches that don't meet these requirements, RAD will fall back to the strawman approach and audit the whole program. As we will detail in Section 3, we inspected 36 patches for Apache and only 2 do not satisfy these requirements.

3. FEASIBILITY STUDY

This section presents a feasibility study of RAD's approach based on a visual inspection of reported vulnerabilities and the corresponding patches for the Apache web server. Figure 3 summarizes all 36 vulnerabilities announced in every Apache 2.2.x release [1], from 2005 to 2010. These vulnerabilities are located in Apache's core code, modules, lower-level libraries (APR and APR-util), and third-party libraries (expat). We determine whether RAD would be able to detect these vulnerabilities correctly or report false alarms by manually inspecting their patches.

12 vulnerabilities (marked as ∅) do not require auditing because they don't result in a compromise but just cause denial of service (i.e., crash or hang). Particularly, 5 may lead to null pointer dereference, 4 may consume more memory than necessary, 2 may hang the server, and 1 may cause the "billion laughs" attack [4], i.e., infinite recursion. None of these vulnerabilities allow attackers to break the integrity of the system or sniff any sensitive information.

Out of the 24 other vulnerabilities, we believe RAD would catch exploits and incur no false alarms for 15 of them (marked as ✓). Their patches meet the requirements discussed in Section 2.2. Section 4 uses two of them for a detailed case study.

For the remaining 9 vulnerabilities, RAD may report false alarms (marked as ☒). We further break them down into the following categories.

Memory layout (3). Figure 4 shows one of the cases, CVE-2008-2939. The patch allocates a new memory block to hold the sanitized input, thus the memory layouts diverge in the two executions, though they are structurally equivalent. RAD's naïve diff of writes would report false alarms for inputs that don't contain any dangerous characters.

Character encoding (3). The browser may be tricked into interpreting the HTTP response using an incorrect character encoding (e.g., UTF-7), if the server does not set it explicitly. The attacker can exploit this vulnerability using carefully chosen inputs to mount a cross-site scripting (XSS) attack. The patch enforces the encoding at the server side, and results in a change of every response, even where there is no compromise. Thus, RAD would report false alarms.

Web page (1). An administration web page provided by one of Apache's modules is vulnerable to cross-site request forgery (CSRF) attacks. The patch adds a token to the forms in the web page, which

Version	Identifier	Component	Type	Detectability
2.2.17	CVE-2009-3720	expat	buffer overread	✓
	CVE-2009-3560	expat	buffer overread	✓
	CVE-2010-1623	APR-util	resource exhaustion	∅
2.2.16	CVE-2010-2068	`mod_proxy_http`	logic error	✓
	CVE-2010-1452	`mod_cache` & `mod_dav`	null dereference	∅
2.2.15	CVE-2010-0425	`mod_isapi`	logic error	✓
	CVE-2010-0434	`mod_headers`	dangling pointers	✓
	CVE-2010-0408	`mod_proxy_ajp`	hang	∅
2.2.14	CVE-2009-3094	`mod_proxy_ftp`	null dereference	∅
	CVE-2009-3095	`mod_proxy_ftp`	missing checks	✓
	CVE-2009-2699	APR	hang	∅
2.2.13	CVE-2009-2412	APR	integer overflow	✓
2.2.12	CVE-2009-1890	`mod_proxy`	resource exhaustion	∅
	CVE-2009-1191	`mod_proxy_ajp`	logic error	✓
	CVE-2009-1891	`mod_deflate`	resource exhaustion	∅
	CVE-2009-1195	config	logic error	☒ design
	CVE-2009-1956	APR-util	off-by-one	✓
	CVE-2009-1955	APR-util	billion laughs	∅
	CVE-2009-0023	APR-util	heap underwrite	✓
2.2.10	CVE-2010-2791	`mod_proxy_http`	logic error	✓
	CVE-2008-2939	`mod_proxy_ftp`	cross-site scripting	☒ memory
2.2.9	CVE-2007-6420	`mod_proxy_balancer`	cross-site request forgery	☒ web page
	CVE-2008-2364	`mod_proxy_http`	resource exhaustion	∅
2.2.8	CVE-2008-0005	`mod_proxy_ftp`	cross-site scripting	☒ charset
	CVE-2007-6422	`mod_proxy_balancer`	null dereference	∅
	CVE-2007-6421	`mod_proxy_balancer`	cross-site scripting	☒ memory
	CVE-2007-6388	`mod_status`	cross-site scripting	☒ memory
	CVE-2007-5000	`mod_imagemap`	cross-site scripting	☒ charset
2.2.6	CVE-2007-3847	`mod_proxy`	buffer overread	✓
	CVE-2006-5752	`mod_status`	cross-site scripting	☒ charset
	CVE-2007-3304	MPM	SIGUSR1 killer	☒ design
	CVE-2007-1862	`mod_cache`	information leak	✓
	CVE-2007-1863	`mod_cache`	null dereference	∅
2.2.3	CVE-2006-3747	`mod_rewrite`	off-by-one	✓
2.2.2	CVE-2005-3357	`mod_ssl`	null dereference	∅
	CVE-2005-3352	`mod_imagemap`	cross-site scripting	✓

∅ = auditing not required ✓ = RAD works ☒ = false alarms

Figure 3: Vulnerabilities in Apache 2.2.x releases [1].

changes every output, even when there is no compromise. RAD would report false alarms.

Design flaw (2). Two vulnerabilities involve design flaws. A malicious local user may override some permissions enforced in `httpd.conf`, or kill an arbitrary process by spoofing the scoreboard. The patches change either the semantics or the architecture of the Apache program. Whole-process auditing seems unavoidable since the changes are global, instead of local to a single function. For Apache, whole-process auditing can lead to false alarms due to non-determinism.

In summary, RAD may report false alarms for 9 of the 36 vulnerabilities in Apache. Specifically, 3 XSS cases require a more elaborate diff, 4 charset-XSS and CSRF cases need further incorporation with browsers, and the 2 design cases would require whole-process auditing. Section 5 speculates how we might handle these cases.

4. INITIAL RESULTS

This section presents the results of applying RAD to retroactively auditing two vulnerabilities: CVE-2009-0023 in Apache 2.2.10 and CVE-2005-3352 in Apache 2.2.0. For these two vulnerabilities RAD does not report false alarms for normal workloads, and is effective in detecting all exploits of the vulnerabilities. All the experiments are conducted on 64-bit Ubuntu 10.10 with Linux kernel 2.6.35.

4.1 Case study I: Heap underwrites

The first case study of retroactive auditing is CVE-2009-0023, the running example shown in Figure 2. The vulnerable function `apr_strmatch_precompile` is invoked by both the server core and several modules, such as `mod_substitute`, which uses it to perform string substitutions on HTTP response bodies. This vulnerability allows a malicious local user to mount an attack by creating an `.htaccess` configuration file that contains a bad string.

To evaluate whether RAD will generate false alarms we create two different `.htaccess` files and enable `.htaccess` in `httpd.conf`. The first file is created in a directory named `good`, which contains the following line:

```
Substitute s/work/sink/n
```

```
--- modules/mappers/mod_imagemap.c ...
+++ modules/mappers/mod_imagemap.c ...
@@ -342,6 +342,6 @@
 if (!strcasecmp(value, "referer")) {
   referer = apr_table_get(r->headers_in, "Referer");
   if (referer && *referer) {
-    return apr_pstrdup(r->pool, referer);
+    return apr_escape_html(r->pool, referer);
   }
```

Figure 5: Part of the patch for CVE-2005-3352, cross-site scripting in mod_imagemap.

A web page in this directory that originally displays "it works" will be rewritten to "it sinks", where the string "work" is used as input to `apr_strmatch_precompile`.

To simulate an attack we create another `.htaccess` file in a directory named `bad`; this file differs from the previous one in changing the character "o" in "work" to `0xf0`. This string tricks the vulnerable function into writing integer 2 into `shift[-16]` rather than `shift[240]`.

To generate a workload, we send 100 HTTP requests for files in the `good` directory. RAD records 110 invocations of `apr_strmatch_precompile`, among which are 100 calls from `mod_substitute`, and the remaining 10 are invoked from the server core and other modules. RAD audits each invocation and reports no warnings.

We also request one file in the `bad` directory. This time RAD reports one alarm and outputs the following diff of memory addresses and corresponding hex values:

```
            address  original  patched
0x00000000007ac188         02       --
0x00000000007ac189         00       --
0x00000000007ac18a         00       --
0x00000000007ac18b         00       --
0x00000000007ac18c         00       --
0x00000000007ac18d         00       --
0x00000000007ac18e         00       --
0x00000000007ac18f         00       --
====================================
0x00000000007ac988         04       02
```

The diff indicates that the vulnerable code wrote the value 2 to address 0x7ac188 (`shift[-16]`), which is the heap underwrite. The patched code did not write to this address. It also shows that the vulnerable code left 4 at 0x7ac988 (`shift[240]`), while the patched code wrote 2 there.

4.2 Case study II: Cross-site scripting

Figure 5 shows part of the patch for CVE-2005-3352, a vulnerability in `mod_imagemap`, a module for processing imagemap files. The vulnerable code copies the value of the "Referer" field to the HTTP response without sanitizing it, which allows attackers to launch cross-site scripting attacks. The patch escapes the value. Note that unlike the previous cross-site scripting case in Figure 4, in this case both the vulnerable and the patched code return a duplicated string, so the memory layout is unchanged by the patch.

To test RAD, we create an imagemap file served by the vulnerable Apache server, and set up a separate web site, hosting a normal web page `good.html`, and `bad.html`, which will exploit the vulnerability to steal a user's cookie. As a user, we visit both web pages using the IE browser from a Windows XP SP3 machine.

In the first experiment, a user visits `good.html` 10 times. RAD records 80 invocations to the vulnerable function in `mod_imagemap`, and audits each invocation by comparing the memory writes by the vulnerable and the patched code. It reports no warnings.

In the next experiment, the user visits `bad.html`. RAD audits the invocations to the vulnerable function, and reports one alarm. The important excerpt from the memory diff between the vulnerable and patched runs is as follows:

```
            address  original  patched
0x0000000001da2b11         >        &
0x0000000001da2b12         <        g
0x0000000001da2b13         s        t
0x0000000001da2b14         c        ;
0x0000000001da2b15         r        &
0x0000000001da2b16         i        l
0x0000000001da2b17         p        t
0x0000000001da2b18         t        ;
```

This excerpt shows part of the injected `<script>` tag that was passed through by the original code, and the corresponding escaped characters from the patched code.

5. FUTURE CHALLENGES

If an attacker exploits a vulnerability, RAD will identify it. The main challenge for RAD is avoiding false alarms. The preliminary experimental results for fine-grained auditing of two vulnerabilities are promising: RAD does not report any false alarms and identifies compromises correctly. The feasibility study further suggests that RAD should do well on many vulnerabilities: 15 out of 24 should not cause any false alarms.

To reduce false alarms for the remaining ones, we plan to refine the diff approach by comparing data structure topology, rather than the raw diffs of memory writes. We can consider two memory images to be equivalent if the corresponding graphs formed by memory blocks and pointers among them are equivalent, i.e., isomorphic, even if the addresses of memory blocks differ. This plan should eliminate false alarms caused by the XSS patches, as long as the patches do not introduce or remove any pointer aliasing (which would make the memory graphs non-isomorphic).

Another source of false alarms arises from client-side exploits, such as the charset-XSS and CSRF vulnerabilities. To determine whether they actually caused attacks at client side, we would like to integrate the browser behavior into the auditing system. For example, RAD could see whether the browser generates the same DOM tree for the HTTP responses from the web server with and without the patch.

In addition to memory differences, RAD should also record system calls issued during re-execution to determine if there are differences in the effects on the execution environment. To avoid conflicting changes to the system state, we expect that RAD would allow system calls from one of the forked processes (e.g., the unpatched parent) to proceed, and intercept system calls from the other fork (e.g., the patched child). If the child's system calls or arguments differ from those in the parent process, an alarm is raised and the child is terminated. If the child's system calls and arguments are the same, the return values from the parent's corresponding system calls are supplied to the child.

Finally, we would like to extend RAD to help the administrator determine exploit severity and analyze actual damages (e.g., determine which files an adversary has modified). We plan to leverage Retro [7] to address both issues.

6. RELATED WORK

Existing intrusion detection tools include COPS, Snort, and Tripwire [6], among many others [9]. These tools log system activities and report anomalies according to a set of predefined security rules. RAD takes a different approach: it uses patches to detect exploits of software vulnerabilities.

IntroVirt [5] asks developers to write predicates in addition to patches for every vulnerability. IntroVirt then detects past intrusions

by checking these predicates while replaying past executions of the system at the virtual machine level. For example, the predicate for the vulnerability in Figure 2 would be as follows:

$$\begin{aligned} &\text{For any character } c \text{ in } \texttt{s}, \\ &\qquad 0 \le c \le 127 \quad \text{if } \texttt{case\_sensitive} \text{ is true,} \\ &0 \le \texttt{tolower}(c) \le 127 \quad \text{otherwise.} \end{aligned}$$

RAD's new auditing scheme leverages patches to remove the burden of writing predicates.

Retro [7] uses re-execution to repair the system. It asks the administrator to mark the initial intrusion point, and re-executes legitimate actions to construct a new system state as if the intrusion never happened. RAD can use a system like Retro to roll back to an earlier state and replay inputs. It extends Retro by retroactively identifying compromises based on security patches. If RAD finds an attack, an administrator can then use Retro to recover.

Delta execution [11] is a mechanism to validate untrusted code changes by comparing system behaviors running different versions of the program. In contrast, RAD trusts security patches, and uses them to detect intrusions.

BinHunt [3] displays the differences between the original and the patched versions of a program by constructing their control flow graphs from instructions and comparing the two graphs. RAD further tells whether the two versions would behave the same given a particular input, and uses that information to infer whether a security vulnerability may have been exploited.

7. CONCLUSION

RAD allows an administrator to determine whether an attacker exploited a vulnerability. RAD audits the past execution of a system based on security patches released by vendors. It retroactively applies the patch and executes the patched code with the original input, and compares the resulting output with the original output. If the outputs are different, RAD concludes that the vulnerability may have been exploited. A feasibility study suggests that this approach is applicable to a complex program like the Apache web server.

Acknowledgments

We thank the anonymous reviewers for their feedback. This research was partially supported by the DARPA Clean-slate design of Resilient, Adaptive, Secure Hosts (CRASH) program under contract #N66001-10-2-4089, and by NSF award CNS-1053143. The opinions in this paper don't necessarily represent DARPA or official US policy.

References

[1] Apache httpd 2.2 vulnerabilities. `http://httpd.apache.org/security/vulnerabilities_22.html`.

[2] B. Chelf and A. Chou. Controlling software complexity. `http://www.coverity.com/library/pdf/ControllingSoftwareComplexity.pdf`, 2008.

[3] D. Gao, M. K. Reiter, and D. Song. BinHunt: Automatically finding semantic differences in binary programs. In *Proceedings of the 10th International Conference on Information and Communications Security*, Birmingham, UK, October 2008.

[4] E. R. Harold. Tip: Configure SAX parsers for secure processing. `http://www.ibm.com/developerworks/xml/library/x-tipcfsx.html`, 2005.

[5] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, Brighton, UK, October 2005.

[6] G. H. Kim and E. H. Spafford. The design and implementation of Tripwire: A file system integrity checker. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, Fairfax, VA, November 1994.

[7] T. Kim, X. Wang, N. Zeldovich, and M. F. Kaashoek. Intrusion recovery using selective re-execution. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation*, Vancouver, Canada, October 2010.

[8] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Chicago, IL, June 2005.

[9] D. Swan. comp.os.linux.security FAQ. `http://www.linuxsecurity.com/docs/colsfaq.html`, 2002.

[10] L. Torvalds. Re: [RANT] Linux-IrDA status. `http://lkml.org/lkml/2000/11/8/1`, 2000.

[11] J. Tucek, W. Xiong, and Y. Zhou. Efficient online validation with delta execution. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, Washington, DC, March 2009.