HYDRA- Hypothesis Driven Repair Automation¹

Full Paper[†]

Brett Benyo

Raytheon BBN Technologies
10 Moulton Street, Cambridge, MA
USA
Brett.Benyo@Raytheon.com

Shane Clark

Raytheon BBN Technologies
10 Moulton Street, Cambridge, MA
USA
Shane.Clark@Raytheon.com

Aaron Paulos

Raytheon BBN Technologies
10 Moulton Street, Cambridge, MA
USA
Aaron.Paulos@Raytheon.com

Partha Pal

Raytheon BBN Technologies
10 Moulton Street, Cambridge, MA
USA
Partha.Pal@Raytheon.com

ABSTRACT

HYDRA is an automated mechanism to repair code in response to successful attacks. Given a set of malicious inputs that include the attack and a set of benign inputs that do not, along with an ability to test the victim application with these labelled inputs, HYDRA quickly provides rank ordered patches to close the exploited vulnerability. HYDRA also produces human-readable summaries of its findings and repair actions to aid the manual vulnerability mitigation process. We tested HYDRA using 8 zero-days, HYDRA produced patches that stopped the attacks in all 8 cases and preserved application functionality in 7 of the 8 cases.

CCS CONCEPTS

• Security and Privacy → Software and Application Security;

KEYWORDS

Automated software repair, resiliency, zero-day vulnerability

ACM Reference format:

B. Benyo, S. Clark, A. Paulos and P Pal. 2018. HYDRA- Hypothesis Driven Repair Automation. In *Proceedings of ACM ARES conference, Hamburg, Germany, August 2018 (ARES'18)*, 10 pages.
<https://doi.org/10.1145/3230833.3230861>

1 INTRODUCTION

Zero-day vulnerabilities are a severe threat to deployed systems, especially those that serve clients over a public network like the Internet. The complexity of modern software exacerbates the zero-day problem. A *simple* web application relies on complex interacting layers such as Apache httpd, PHP, MySQL, bash, OpenSSL, and the OS kernel. Exploitable vulnerabilities can exist in any of these layers, and since 2013, attackers have increasingly used zero-day exploits with a 64% increase in 2013 from 2012 [17].

As an example of a zero-day vulnerability, consider CVE 2012-1823 [19], which allowed an attacker to execute arbitrary PHP code on an Apache httpd server using a specific but standard PHP configuration. This bug was live for 8 years. The initial disclosure was secret, and 4 months passed with no patch, until a bug report inadvertently leaked. Exploits became immediately available. The first official patch to the PHP source code was incomplete, and vulnerable to a slightly modified attack that appeared in the wild immediately. A second patch, released five days later, finally closed the vulnerability. It is sobering to think that an attacker could have exploited any Apache server with a zero-day attack for 8 years. CVE-2012-1823 is just one disclosed vulnerability; it is impossible to determine how many others remain undiscovered in commonly used software stacks today.

In 2014, *Shellshock* (CVE-2014-6271 [20]) made national headlines as system administrators scrambled identify and patch vulnerable applications. Shellshock was very widespread, with 173 million attacks against Akamai customers alone [18], and was live

¹ Approved for Public Release, Distribution Unlimited.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

ARES 2018, August 27–30, 2018, Hamburg, Germany

© 2018 Association for Computing Machinery.

ACM ISBN 978-1-4503-6448-5/18/08...\$15.00

<https://doi.org/10.1145/3230833.3230861>

for decades. Once again, initial patches were incomplete, leaving patched systems vulnerable to modified attacks.

As Shellshock [20] and Heartbleed [25] have demonstrated, it is not a matter of if, but when, a given software stack will suffer a zero-day attack. The mean lifetime of an exploited zero-day vulnerability is around 10 months [31], and even after public disclosure to maintainers, patch development can take weeks. Many vulnerabilities require multiple patch cycles. These delays are unacceptable for critical systems, which cannot tolerate precautionary downtime or incomplete mitigations.

Irrespective of what the ultimate solution to the zero-day problem can be (e.g., proven bug-free software or approaches to eliminate large classes of vulnerabilities), there is an immediate need for a defense of last resort for deployed critical applications. Long before RASP (Runtime Application Self Protection) appeared in the security lexicon [32], research systems like A3 [21][22][23] demonstrated managed execution environments with hooks for monitoring and recording application inputs and state, and introspection and playback of recorded inputs. These environments opened up the possibility of automating post-incident responses such as restoring service from a past state, isolating offending inputs to a small set, and deriving patches for the successful exploit.

In this paper, we introduce HYDRA (Hypothesis Driven Repair Automation) – a mechanism to provide fast, automated code repair patches in response to successful zero-day attacks within a RASP or managed environment like A3. In addition to locally mitigating an exploited vulnerability so that the deployed application can operate with improved protection while the regular patch cycle takes its time, HYDRA also provides suggestions and summaries that human experts can leverage to get to the generally applicable fixes faster. HYDRA makes the following contributions:

- Automated localization of attack exploited vulnerable code.
- Automated code modification and testing to mitigate the localized vulnerability without compromising functionality.
- Human-intelligible context about the exploited vulnerability, candidate patches and patch quality to aid manual patching.

We tested HYDRA's automated repair on exploits of 8 zero-day vulnerabilities. HYDRA found repairs that prevent the attack in all 8 cases, 7 of which preserved all relevant application functionality.

2 Background

HYDRA uses test cases that exhibit anomalous behavior to start its patching process. These test cases can be developer-provided tests like those used by GenProg [1] and RSRepair [5], or malicious inputs that reproduce the attack effects. Our focus in this paper is repairing live network facing server applications protected by RASP or managed execution environments like A3. In this context, HYDRA uses playback of malicious traffic that deterministically reproduce failures to trigger patch development, and developer provided and available regression tests for patch validation. The managed environment provides the scaffolding and support services required for automated analysis of execution traces and log data, and automated formulation and testing of hypotheses about exploited code and potential fixes. Since A3 is the deployment and

evaluation context of HYDRA, we describe A3 next as a concrete example of a representative operating context for HYDRA.

A3 [21][22] is a managed execution environment designed to detect and mitigate the impacts of zero-day attacks. With A3, a protected application provides service from inside a set of virtual machines (VMs) managed by the A3 controller. The A3 controller is responsible for monitoring the resident VMs for evidence of successful attacks, recovering the application to a previous state in the event of an attack, and running traffic replay experiments to generate network filters that prevent the attack in the future. A3 allocates two sets of VMs to allow its diagnostic and repair automation to run in parallel with the deployed application, as depicted in Figure 1. The VMs in the *Production* area provide service to clients. The VMs in the *Experiment* area are used for analyzing the application's execution through replay experiments, and testing mitigations by regression and traffic replay tests. A3's principal monitoring mechanisms are monitoring of I/O, and Stackdb library [29] based Virtual Machine Introspection (VMI).

Figure 1: HYDRA and the A3 execution environment.

A3's I/O and VMI-based monitoring can detect both application-specific undesired conditions (e.g., this Apache instance should not make outbound calls to this IP range) and generic failures (e.g., abnormal exit codes). When a novel attack bypasses existing defenses and successfully exploits a vulnerability that results in a monitoring policy invariant violation, A3 executes the workflow illustrated in Figure 2. It first restarts the application running in the production area from the most recent checkpoint in an attempt to revert the application to an uncompromised state. In

Figure 2: HYDRA in the post-incident workflow enabled by A3

parallel, in the experiment area, A3 uses replay experiments to isolate the set of network requests sufficient to reproduce the invariant violation, partitioning the inputs recorded for the period before the attack into a *benign* and a *malicious* set. At this point, A3 can choose from the patch generation options (as shown by the dashed boxes on experiment side of Figure 2) that are available to it. For example, prior work has shown how to create a regular expression based network traffic filter using a supervised learning approach. Similarly, mechanisms to explore patches to other security policies are also feasible. While a successful network filter or a security policy patch will block some attempts to exploit the same vulnerability, the vulnerability remains in the code, and a modified attack may bypass the patched network filter or security policy. HYDRA's goal is to identify and directly patch the faulty code that enables the attack's success. In this process, HYDRA leverages a number of basic post incident recovery and response mechanisms provided by A3 such as attack effect detection, isolation of recorded inputs into benign and malicious sets, and replay-based experimentation. After the detection of an application attack or failure, HYDRA uses the malicious partition of the recorded requests and responses and the detected application failure (e.g., `do_exit` with a `SIGKILL`) as a test case to reproduce a failure or determine the absence of a failure after a repair attempt.

HYDRA's code repair is similar in nature to the repair performed by ClearView [2]. ClearView uses Daikon [30] to detect invariant violations that trigger the repair, whereas HYDRA's repair is triggered by the managed execution environment's error detection mechanisms. More on this in Section 6.

3 HYDRA

HYDRA examines execution traces to generate hypotheses about the faulty code segments that enable attacks, tests these hypotheses, and then generates repair candidates to mitigate the fault. HYDRA's analyses fall into two categories. **Test panels** are analysis routines that gather information about an application's execution in response to given inputs, allowing HYDRA to further localize faulty code and identify differences in execution between benign and malicious cases using A3's built-in monitoring. **Hypothesis tests** check whether a specific code modification successfully mitigates the fault under consideration. HYDRA packages the application of a test panel or testing a repair hypothesis as an experiment, executed in the experiment VMs.

HYDRA uses a heuristic-based analysis of (a) the likelihood of a test panel generating high confidence hypotheses, and (b) the likelihood that the repair posed by a repair hypothesis is successful, to determine the utility of an experiment. HYDRA continuously executes the highest utility experiment available until it finds a *suitable repair* or exhausts all potential hypotheses. The hypotheses used in a typical incident response numbered in the tens in our experience, enabling exhaustive search of the hypothesis space.

HYDRA tests the *suitability* of a repair by replaying a set of benign input messages, the full set of malicious input messages, and any existing regression tests. A suitable repair is one that passes all available regression and replay tests, and the malicious input

results in a standard error message or a standard output response, without the original undesired condition.

HYDRA's repairs take the form of conditional or unconditional disabling of a function call or code block, or setting a function call parameter or local variable to a specific value. These repairs may degrade the application, causing some regression tests to fail. However, in many cases the reduced functionality is unused by the application in the context in which it is operating, and serves only as an attack vector. If source code and a build environment are available, HYDRA modifies the application code, synthesizing the potential code modifications from a small set of repair templates. In cases where the application build environment is unavailable, HYDRA produces VMI repair candidates that use the Stackdb VMI library to alter application execution in the same way that a code modification patch might. The VMI repairs pause the execution of the managed application at the point of a matching system call or user space function, and prevent the call from proceeding if parameters match the regular expression specified in the VMI repair. HYDRA prevents the calls by immediately returning to the next function on the call stack, sending the application a signal, or terminating the application. VMI repairs incur high overhead, but work without the source code or build environment.

HYDRA can run in a fully automated mode (the focus of this paper) that begins at fault localization and ends with a patched application, or run in a mixed-initiative mode, where a system administrator can interact with the experiment queue, add, remove, or modify experiments, or change the order of experiments. The latter allows a human expert to leverage her insight and creativity to assist with the repair and gain confidence in the generated patch.

One complexity that arises when modifying source code, whether done by a human or HYDRA, is unintentional introduction of new vulnerabilities. Even though HYDRA repairs reduce application functionality, new vulnerabilities could result, if, for example, the disabled executions take out conditional guards that modify the control flow. As an example, if the vulnerability was in the clean-up code at the end of a function and HYDRA circumvents it with an early return statement, a memory leak could ensue. Ideally, the full removal of such a conditional would cause a failure in the regression tests. However, regression tests are rarely so thorough in practice. HYDRA's mixed-initiative approach is a potential mitigation, since HYDRA's explanation enables a human developer to understand and analyze the proposed repair for unintended side effects. HYDRA's repairs are small, usually single lines of code, and thus easier for a human to evaluate. Finally, newly introduced vulnerabilities will take time for attackers to find and exploit, buying time for expert code review and analysis.

3.1 Test Panels

Test panels are customizable routines that examine a replayed malicious input set using execution tracing with VMI to record new information, such as call stacks or variable values. HYDRA uses test panel results (i.e., observed patterns or conditions) to trigger hypothesis generation. This is analogous to a human analyst inspecting logs or tracing the application's execution, to watch for a specific event that tests a hypothesis. The configuration of a test

panel, the conditions to look for, hypotheses to generate, and additional experiments to run are predefined. However, the test panel infrastructure is extensible, enabling engineers to enhance test panels as they gain more experience with HYDRA. We describe the test panels used in the experiments in this paper next.

3.1.1 Probe Statistics. This test panel places a VMI trigger on every system call to record the call and its parameters. The panel then lets the call execute. For attacks that introduce new functionality, this panel can uncover differences in the series of system calls when contrasted with a benign replay. More subtle attacks that do not introduce new calls and tend to reuse application logic may manifest as different parameters to the calls. This test panel may be ineffective for some attacks that minimally manipulate application logic and avoid introducing new system calls or system call side effects. As such, it is prudent to assume that no single test panel is comprehensive by itself.

The repair hypotheses that the probe statistics panel generate uses VMI repairs (i.e., runtime execution manipulation) to prevent detected errant system calls or calls with matching parameters that differ from benign traces from executing. VMI repair can either abort a matched call with a failure return code, or kill the entire process before it executes the call (e.g., SIGKILL). Instead of deciding which to do beforehand, HYDRA can instantiate both hypotheses and test one after the other. If the VMI repair prevents the undesired condition from occurring without causing any of the benign inputs in the regression test to fail, HYDRA schedules further test panels. At the same time, HYDRA can recommend the VMI repair as a temporary mitigation for the live application.

3.1.2 Call Stack Analysis. This test panel further examines the cases where a system call with specific parameters appears only in the execution of malicious inputs. Its objective is to perform an initial step similar to root-cause analysis. It starts by placing a VMI trigger on the system call in question and generates a full function call stack for every call, regardless of parameter values for benign and malicious inputs. The current implementation uses a GDB script running in the application VM to gather this information, but future versions could use Stackdb to generate the call stack from the VMI layer, eliminating the overhead and restrictions of using GDB. If the system call never occurs for benign inputs, we generate repair hypotheses that disable code for each function call in the call stack. If the system call occurs for benign inputs, we match stack frames from the benign cases with the malicious cases by code address. We then gather a set of parameter values and values for local variables on the stack in each stack frame, and compare parameter values and local variable values in matching stack frames. We ignore stack frames where the function call has no associated source file or a file in a location that is marked in HYDRA configuration as not modifiable. For cases where a parameter or local variable has a unique value for malicious stack frames, HYDRA generates a Blacklist Conditional Disable code repair hypothesis. Alternatively, if every benign stack frame has the same parameter or local variable value, it instead generates a Whitelist Conditional Disable code repair hypothesis.

3.1.3 Null Pointer. For the cases where the application exits with a SIGSEGV, we use an extension of the general Call Stack

Analysis test panel, where we generate a stack trace when the signal gets thrown and look through the parameters and local variables for values that are set to 0x0. For these, we generate repair hypotheses using a small set of code templates that insert null pointer checks prior to the function call in the stack frame. These templates either prevent the call if the inspected value is null, or set the null value to a predefined non-null value and continue with the call.

3.2 Code Repair Hypothesis

Code repair hypotheses are instantiated by a test panel, with a line of source code and a repair template to apply to that line. Since having a compilation environment is a requirement for source code repair, we assume that we can generate a binary with full debug symbols available, which allows stack trace map execution to line numbers in source code. For some hypotheses, multiple repair templates may apply. For example, to disable a function call, one could comment out the call or insert a return, backing out of the encompassing function entirely. We currently have a small set of templates, and simply try them all instead of trying to identify a good candidate through static analysis. A code repair template is a few lines of code for insertion at one line in one source file. Generating and compiling the variant is generally very fast. Any candidates that cause a compile error are quickly abandoned. We next describe the HYDRA repair templates.

3.2.1 Blacklist Conditional Disable. This repair template conditionally returns instead of executing a target code block. For example, the template for replacing the function call: `func(a,b)` would be: `if (condition) { return(retval); } else {func(a, b); }`. Condition is a union of parameter or local variable names and values, such as `(argc = 2 && cgi = 0)`. This template has additional variants for the return (`retval`), which is intended to short circuit this code path instead of executing `func`. Other variants try a `continue` call, guessing that this line may be inside a loop, or a `no-op`, which would just prevent `func` from being called, and then `continue` on with the rest of the encompassing function.

3.2.2 Whitelist Conditional Disable. This repair template simply reverses the conditional from the blacklist example, executing `func` only if the condition is met.

3.2.3 Force Variables. This template unconditionally sets variables to have the expected value immediately before the target function call. The assumption for this template is that the vulnerability involves an unintended code path triggered by propagating unexpected values to the target variables. This applies to bugs that amount to incorrect constraint checking. A viable repair candidate produced by Force Variables will likely not fix the root cause of the vulnerability, which may be upstream in the code. However, the repair may neutralize the exploit, provide a workaround capable of getting the application up and running immediately, and give the developers a head start for the real patch.

3.3 Testing Hypotheses and Assessing Patches

To test a hypothesized code repair, HYDRA generates the code variant from a template and executes the compilation script. If compilation produces an error, HYDRA abandons the hypothesis and moves onto the next experiment. If compilation succeeds,

HYDRA runs three types of tests. First, it replays the malicious inputs with the variant binary, testing to see if the application produces the undesired condition. HYDRA also checks for other policy violations, crashes, or pre-defined application errors. Second, HYDRA replays a set of benign inputs, looking for application errors, crashes, or policy violations. In addition, we can add an application-specific test to check if the repaired application handles benign inputs successfully. These tests, created specifically for each application, are very high level and simply check for valid responses with standard non-error status codes. Such an acceptance test may be insufficient in some cases, e.g., if a HYDRA-generated repair generates a response with a standard code, but produces incorrect content or other side effects. Our subjective patch evaluation would flag this, and we would update the test to check for more conditions. In fact, the only case where simple acceptance proved insufficient was with httpd and CVE-2012-0012, which was a vulnerability in the logging method. We added a condition to the httpd acceptance test to check for a properly formed log message, which resulted in an acceptable repair.

Finally, HYDRA runs the regression test suite, and counts the number of passed tests. These benign, malicious, and regression test results are combined into one score through a simple fitness function f , whose parameter are explained in Table 1 (P^i and N^i refer to benign and malicious counts passing the test):

$$f = \sum_1^p (P_i^1 + \alpha P_i^2 + P_i^3) + \sum_1^n (N_i^1 + \alpha N_i^2 + N_i^3) + \sum_1^r R_i \quad (1)$$

Table 1: Parameters of the fitness function

Parameter	Description
p	Number of benign inputs- inputs known to be benign either provided with the application or discovered through record and replay. The tests replay these looking for specific effects.
N	Number of malicious messages- inputs to the application known to be malicious.
R	Count of regression tests provided with the application (optional). For applications with a large number of regression tests, we choose a representative subset. A full regression is optional.
P^1, N^1	Count of benign message (P^1) and malicious message (N^1) passing tests that check for a valid response, defined by the application. i.e. 200 OK for HTTP, or a valid DNS Response for BIND.
P^2, N^2	Count of (benign, malicious) messages passing tests that check absence of undesired condition.
P^3, N^3	Count of (benign, malicious) messages passing additional application-specific checks (optional).
α	Weight given to preventing the undesired condition

One point is awarded for each passed test except for the test of producing the undesired condition where we add an additional factor, currently set to 10, to reflect increased importance in preventing it. Specific details of the fitness function are less important, since HYDRA is not using fitness scores to explore a large search space, but instead to rank hypotheses in a smaller well-defined space. Fitness scores fall into one of the following 6 categories, and we determine which category by examining the individual components in Table 1.

1. Compile Error: No tests pass, $f = 0$
2. No effect: All tests that pass in the base case still pass, and the malicious message(s) still cause the undesired condition, $f \approx 3p + r + 2$ for the tests $N1$ and $N3$, which may also pass depending on the specific vulnerability.
3. Broken: Change broke processing of the benign messages, $f \approx p + \beta + \alpha n$, where $\beta \leq r$. β is the number of passing regression tests. The undesired condition may be gone, so $P2$ and $N2$ tests pass, but likely because input processing is aborted too early or indiscriminately.
4. Reduced Functionality: $f = 3p + 2n + \alpha n + \beta$, where β (the number of regression tests passed) is less than r . All benign messages process properly, the undesired condition does not occur, but some regression tests fail. This can indicate that the failed regression tests check for functionality that is not necessary for processing the types of request the application, or functionality that is less important and not exercised.
5. Malicious Error Response: The malicious input results in either an error response or no response, without producing the undesired condition, and benign functionality is not affected, $f = 3p + \alpha n + r$. Error responses may leak information to the attacker or produce other side effects that were not tested for.
6. Perfect: $f = 3p + (2+\alpha)n + r$. The malicious message is processed normally, does not produce the undesired condition, leaking no additional information to the attacker.

HYDRA uses these categories to guide the search and inform the human administrator. Cases 4, 5, and 6 are all repair candidates since they prevent the undesired condition and preserve the tested functionality of the benign inputs. HYDRA stops the search when it finds a *Perfect* repair as no repair can score higher than *Perfect*.

Ideally, applications would come with full suites of regression tests that cover the entire expected functionality of the applications. However, this is rarely the case. For applications with imperfect regression test coverage, a *Perfect* score can still result in reduced functionality, affecting functionality not covered by a test. Most of the open source widely used applications such as httpd, PHP, and BIND are so feature rich that a given installation rarely uses all features. It is entirely possible that the *reduced* functionality is unused in a given deployment and the repair does not impair its use.

4 Experimental Evaluation

Our evaluation of the HYDRA prototype focuses on the plausibility of the patches that HYDRA proposes. Evaluation of prior work closest to HYDRA such as ClearView repair subsystem focused on whether patches allow the application to keep operating in the evaluation context. We followed a similar approach, however since HYDRA is run in the context of a RASP or managed environment like A3, we did not have to construct an artificial evaluation context- the live deployment environment provides one. Unlike the ClearView evaluation, we have ground truth for our repair attempts in the form of official developer patches, and thus we can compare HYDRA generated patches with the actual patches.

Since our focus in this paper is to defend network-facing servers against zero-days, we used versions of a number of commonly used server applications and recent CVEs they are vulnerable to as

experiment targets. The applications we chose include the Apache httpd web server, the PHP interpreter, the BIND DNS server, and the bash shell. These are popular server programs regularly exposed directly to inputs from the Internet. The choice of bash may seem odd, but Shellshock has shown that servers are vulnerable to bash bugs. We searched published CVEs to identify vulnerabilities to repair. In order for a vulnerability to be a viable candidate for testing, we needed to affect the application from a malicious input, which can require considerable engineering effort because many reported vulnerabilities only came with vague descriptions. We focused our search on vulnerabilities that are potentially exploitable by a remote client. In addition, we had an external red team insert new vulnerabilities into a version of BIND for our testing. Table 2 shows the eight distinct vulnerabilities we used in our evaluation.

4.1 Experiment Design and Setup

To trigger HYDRA's repair, the exploited vulnerability must cause an effect on the defended application that the host environment (which is A3 in our current case) can detect. Application crash, often unintentional, while the attacker wanted a different outcome, is the most common effect of a successful exploit. While crash is easy to detect, some vulnerabilities allow arbitrary code execution, which can cause almost any effect on the victim. For these vulnerabilities, we specifically designed the malicious inputs to cause an effect for which A3 has a detector, such as reading a protected file or a crash stemming from altered code execution. We took this approach because our goal is to evaluate HYDRA, not A3's detection framework. Our chosen applications are open source, written in C, and built on Linux using the autoconf and make utilities. The regression tests varied per application, so we wrote simple scripts to automate test running and result tabulation.

In order to test HYDRA on the eight chosen vulnerabilities, we first gathered a set of benign input messages captured from live client events at an MIT Lincoln Laboratory Capture the Flag Event [27] for httpd, PHP, and bash. For BIND, we captured a series of requests from a military exercise where BIND was part of a client/server application. For each of the chosen vulnerabilities, we generated a malicious input that triggered the vulnerability by reverse engineering the exploit based on publically available exploits and the patched code. For CVE-2011-3607, our malicious input caused a crash instead of remote code execution, since the vulnerability in question was a heap overflow, and achieving code execution is nondeterministic, theoretical (no live code execution exploit has been reported in the wild), and was unnecessary for our purpose. For CVE-2013-4854, we added code to BIND to help trigger the exploit because no exploit was publicly available. For some of the vulnerabilities, we slightly modified our application configuration to make it vulnerable. For example, for Shellshock (CVE-2014-6271), we switched the default shell to bash (from dash), and for CVE-2014-0238, we added a file info column to the web application's table, so that the PHP fileinfo methods are used.

4.2 Metrics

For each vulnerability, we calculated the quality of the best repair candidate based on the fitness score and then subjectively analyzed

the repair by comparing it to an actual patch published by the application developers. The time HYDRA takes to generate these repairs is heavily dependent on the number of replayed benign input tests and the size of the regression test suite. Parallelizing the experimentation (if multiple experiment nodes are available) can further reduce the time to generate a repair. Consequently, we report the number of experiments generated and tested, as opposed to wall clock time. On average, it took ~30 seconds per experiment, leading to overall repair times ranging from minutes to a few tens of minutes. We chose to replay 6 benign inputs as test cases, selected by an Expectation Maximization clustering algorithm based on features extracted from the inputs [22]. We chose the number 6 experimentally; larger sets provided no benefit in our experiments. For the regression test suite, we manually pruned tests that were irrelevant for the application's current deployment. HYDRA can run the entire regression suite and a larger replay test suite after identifying a repair candidate as an additional step.

For the fitness score metric, we use the six categories defined in Section 3.3. For the experimenter's subjective analysis, we use the following 6 categories:

- **Perfect:** The candidate repair is semantically equivalent to the published patch and can be committed to the repository.
- **Needs Refinement:** The candidate repair is in the same function and affects the same code, but is too broad or too specific. A human developer would refine it.
- **Correct Location:** The candidate repair is in the same function as the real patch, but a human developer would likely produce a different repair in the same function.
- **Correct Call Graph:** The repair candidate modified code in the call graph of the method that the actual patch modified.
- **Useful Symptom:** The repair candidate fixed a downstream effect of the real problem, which was repaired elsewhere in the code by the real patch. However, we subjectively determined that the repair candidate would be a useful hint for a developer.
- **Unhelpful:** The repair candidate prevents a downstream effect that has a very weak indirect relationship to the actual bug, and provides a developer with little to no help.

In addition, for non-Perfect repairs, we use experimenter analysis and judgement to determine if the repair candidate is too broad or too narrow. Too broad repairs disable functionality more generally, leaving the application open to false positives (i.e., some benign inputs are not processed correctly). This usually indicates a deficiency in the regression test suite, but in many cases may be irrelevant for the current deployment if live clients never generate such inputs. Too narrow repairs may allow a modified attack input to get through and trigger the undesired effect

5 Experimental Results

Table 2 provides the analysis of the HYDRA generated repairs for our eight chosen vulnerabilities. Four of the eight repairs involve deleting functionality (Disable Code), with two more using a conditional deletion (Blacklist Disable). These six repairs achieve a *Perfect* fitness score: the patched application processes benign messages successfully, the regression tests pass, and the malicious input does not produce the undesired condition. Again, passing all

regression tests does not indicate a true perfect patch, since the regression test suite is never comprehensive. In all six of these cases, the actual (ground truth) repair was either in the same function, call stack, and often involving the same variables. We thus concluded that the proposed repair could provide useful information for human developers to find a generally applicable patch. Of course, these patches allowed the deployed applications to resume service immediately without further exploitation.

For CVE-2012-1823, a functionality deletion did pass the tests but caused a non-standard error message, which could leak information to the attacker. Repairs resulting in non-standard error message have lower scores. HYDRA refines a conditional disable repair by forcing the condition instead of checking for it by changing the value of specific variables to make the condition is true. In this case, the root cause of the vulnerability is upstream, where the variable gets its incorrect value from a bad input. The specific variables and values indicated by this repair is very useful for human experts looking to mitigate the vulnerability.

HYDRA did not find any functionality deletion patch for CVE-2012-0021, since the tests checked for proper access to log output, and the vulnerability was in the log output code. This attack manifested as a null pointer dereference in one of its stages, so HYDRA generated hypotheses to insert null pointer check for all local variables with a null value in the vicinity of the observed null pointer dereference. The final repair used a null check for a variable, and if true returned from the call popping a stack frame.

For CVE-2014-9427, HYDRA was unable to produce a satisfactory patch. This bug was deep in parsing code; HYDRA was able to localize the error to the PHP parser and provided useful guidance to human developers. We describe each vulnerability and the HYDRA generated repair in detail next.

CVE-2014-6271 (Shellshock): This vulnerability was an error in environment variable parsing for function definitions in the bash shell. Bash can be a component of a web server stack, with the web server using a bash script to call, e.g., the PHP interpreter. A system

with a vulnerable bash implementation configured in the right way can allow external input in the HTTP input headers to trigger the vulnerability and execute arbitrary commands on the server. We used an attack input that attempts to exfiltrate the `/etc/passwd` file, triggering a violation in A3's file access policy.

HYDRA started by treating the entire application stack (Apache httpd web server, bash shell, PHP interpreter, mysql database, the underlying OS and standard libraries) as the application to be repaired since the location of the bug that triggered the file access policy violation is unknown. It first used the Call Stack Analysis test panel to gather statistics for all the system calls executed along with call parameters while processing the malicious input, and compared them with all the system calls executed when processing the benign messages. It then generated VMI repair hypotheses for calls or call parameters unique to the malicious input processing.

HYDRA eventually tested a repair hypothesis that blocked any attempt by bash to execute the `sys_clone` system call, since this was a unique system call never used in any benign input processing. Note that the proposed patch is conditional to the entire process hierarchy, and thus only prevents `sys_clone` calls from bash when bash is called from "httpd" through the "PHP-cgi-wrapper" script. This VMI policy repair is successful at preventing the undesired condition (protected file access). Since HYDRA had access to the bash source and compile scripts, it generated and tested a source code repair hypotheses as well.

The actual patch provided by bash developers was in a function a few layers deeper than the HYDRA repair, but in the same call stack. The actual patch modified the culprit parsing routine in a semantics aware manner that is well outside the scope of HYDRA's repair templates. HYDRA generated the patch in roughly 3.5 minutes from the moment A3's file access violation was detected. This quick repair allowed the application to continue running while human developers worked on a refined patch. The test case that triggers the bug and localization of the repair to the correct binary and call stack is also very helpful to the human developers,

Table 2: Results summary. Experiments refer to the number of total experiments HYDRA executed.

Vulnerability	Application	Experiments	Code Repair	Fitness Score	Subjective Analysis
CVE-2014-6271	bash	8	Disable Code	Perfect	Correct Call Graph (Too Broad)
CVE-2012-1823	PHP	14	Whitelist Force Variables	Perfect	Correct Call Graph (Too Narrow)
CVE-2014-0238	PHP	4	Disable Code	Perfect	Useful Symptom
CVE-2011-3607	httpd	2	Disable Code	Malicious Error Response	Correct Location
CVE-2012-0021	httpd	5	Null Pointer Check	Perfect	Needs Refinement
CVE-2014-9427	PHP	34	Blacklist Disable	Broken	Useful Symptom
CVE-2013-4854*	BIND	7	Blacklist Disable	Perfect	Correct Call Graph (Too Broad)
CVE-2015-5477	BIND	6	Disable Code	Perfect	Correct Call Graph (Too Broad)

CVE-2012-1823: The PHP interpreter has command line parameters that allow additional PHP files to be included (or downloaded from a URL) before the target PHP file. The default configuration for running PHP from Apache using cgi has this functionality disabled. CVE-2012-1823 allows an attacker to send arbitrary command-line parameters to the PHP interpreter through cgi. By exploiting this vulnerability, the attacker can re-enable the functionality to download and execute additional PHP files, and force download and execution of a PHP file of his choice.

The HYDRA repair for this vulnerability started with the Call Stack Analysis test panel, which looks for system calls and call parameters unique to the malicious case. After testing and discarding a few non-deterministic call parameter values, HYDRA discovered that the `sys_open` call had a unique value for the filename parameter in the malicious case. Blocking this call when the file name parameter value matches the unique value is not a good patch since the attacker could simply change the filename to download. Therefore, HYDRA continued its search, this time with the Probe Statistics test panel to explore who called `sys_open`, both in the benign as well as malicious cases, and the parameters and local variables in the calls. HYDRA observed that all seven of the `sys_open` calls that occur for every benign input had different values for the filename parameter to `sys_open`. This lowers confidence that the filename parameter should be part of a conditional block. Looking at the local variables, HYDRA observed that the `prepend_file` local variable was always null for the benign inputs but had a non-null value for the malicious case. This variable stores a file descriptor for a PHP file to execute before the original requested file. HYDRA then tested repair hypotheses related to this variable. First a conditional disable source code repair was attempted, which adds an “if” conditional that only executes the function that eventually does the `sys_open` call if `prepend_file` is null. This repair prevented the undesired condition and passed all benign and regression tests, however the malicious input resulted in a non-standard error message, in this case, a blank page. HYDRA tried a Force Variables repair next which added code to reset the `prepend_file` variable to null before executing the next function. This resulted in the standard response for the malicious case (the prepended file is ignored), the regular requested file is processed and a response returned.

For this repair, the actual developer patch fixed the input processing higher in the call stack, but again this repair was in the right location, could have provided useful guidance to the developers, and fixed the vulnerable application in around 6 minutes, allowing it to continue serving live requests.

CVE-2014-0238: This PHP interpreter vulnerability is in the code that parses a rarely used file format (CDF) to return file metadata. A specially crafted CDF file can cause a crash in the PHP interpreter when the `fileinfo` function parses the file. Vulnerabilities causing a crash are of low severity to the httpd and PHP stack, since crashed process is restarted and other threads continue to serve benign requests. However, this vulnerability is more severe for our application since the crash takes out the file upload functionality.

HYDRA started looking for repairs to this vulnerability with the Call Stack Analysis test panel to capture the call stack after the PHP

interpreter exits with a segmentation violation signal. It then generated code disable hypotheses for each call on the stack, in an attempt to disable potentially vulnerable unnecessary functionality. If any of these code disable hypotheses are successful, HYDRA would then look for local variables and function call parameters that are set to null, and hypothesize a conditional disable based on any null variables, guessing that a null pointer dereference may be the cause. For CVE-2014-0238, a code disable for the `file_info` call in CDF presents a *Perfect* scored repair since there is no benign or regression test that involved the CDF file format for this application. Again, this repair is not what a PHP developer would prefer, since the root cause of the error is in the CDF header parsing. However, the generated repair fixes the application at hand and again provides correct localization guidance to the developers.

CVE-2011-3607: CVE-2011-3607 is a complex vulnerability in Apache httpd involving multiple steps and a specific server configuration, resulting in an integer overflow and ultimately a heap buffer overflow. It is theoretically possible to get code execution if the attacker can configure or spray the heap through another method. For the HYDRA experiments, we pre-configured the website so that it was vulnerable by enabling `mod_setenvif`, and assuming that an insider could upload a specially crafted `.htaccess` file. HYDRA does not yet have a test panel to generate specific repair hypotheses for integer overflows. However, our malicious input did cause a crash (of a child process) that led to a minimal service disruption since worker threads in other child processes continued to handle benign requests while the crashed child process restarted. Since the attacker could exploit the side effects of the crash further (i.e., injecting into the heap), repairing it is still time critical and valuable to the live deployment.

HYDRA employed the Call Stack Analysis test panel, because the detected violation is a `do_exit` call. It hypothesized code disable repairs, which immediately fixed the problem, since there is no benign use of the `setenvif` module in this context. The disabled code is just a few lines away from the integer overflow, and a potential future test panel that examines local variable values in the functions involved in the `do_exit` call stack, looking specifically for widely divergent integer values (when compared to benign calls) may be able to generate a more refined patch. In this case, the malicious input returns a non-standard error (no response) instead of the requested page, giving the attacker some information that the attack caused an effect, however this effect is not further exploitable.

CVE-2012-0021: This *cookie crash* vulnerability was a bug in the Apache httpd logging module that caused a crash if the server was configured to log cookie values and a specially crafted input was received. A crash in httpd is not very disruptive, however, in this case the crash prevents the input to appear in the main access log that is very useful forensics. Thus, an attacker could use this vulnerability to mask other traffic or exploits.

Starting with the Call Stack and Probe Statistics test panels, HYDRA looked for local variables that are null in the call stack because this was a null pointer exception. A conditional disable repair that checks for a specific null value in the “name” local variable was able to prevent the crash. When HYDRA generated a conditional disable, it first tried to simply skip the function call in

question with an *if* statement. In this case, that repair resulted in an infinite loop, because the skipped statement had a side effect of incrementing the loop iterator variable. HYDRA timed out when doing replay tests for this repair, giving it a very low score. HYDRA then tried another solution, which disabled the function call with a return statement if the condition is met. This prevented the crash and processed all benign requests without error.

The actual patch uses a very similar condition, but instead of returning, continues to the next loop iteration while properly incrementing the loop iterator. HYDRA localized the repair to the exact function and variable in question. An enhanced version of HYDRA could attempt a continue statement, and with static code analysis, could check that the loop iterator is incremented. This would enable a truly perfect automated repair for this specific case.

CVE-2014-9427: This vulnerability causes a crash in the PHP interpreter when parsing an improperly formatted input file (starting with a # and no newline). HYDRA performed the worst on this vulnerability, since the error is deep in the lexer, and disabling any part of the parser quickly stops processing of any benign input. HYDRA's analysis was still useful- it localized the error to the PHP parser and A3 provided an easily reproducible test case that causes the error. A future HYDRA test panel that adds breakpoints to benign replays and gathers more statistics on the values of local variables may lead to a viable patch.

CVE-2013-4854: This vulnerability is in the BIND DNS server, where a specially crafted DNS request causes a crash, leading to a denial of service. We used a synthetic version of this vulnerability for testing because a malicious input recreating the effect was not publically available. Instead of engaging in exploit development, we added code to BIND to trigger the crash by forcing the generation of a log message. While HYDRA might eventually produce a repair that simply removes the code we added, it first attempted repairs at earlier stack frames, deeper into the code that writes the specific data structure that had the vulnerability. HYDRA was able to produce a repair that conditionally disabled the logging deeper in the logging code. The final developer patch was in the same location, but had a more thorough condition that prevented the false positives we saw in our benign message set.

CVE-2015-5477: This vulnerability was also in the BIND DNS server, and again caused a crash while handling a specially formatted DNS request. CVE-2015-5477 was publicized days before we were to present a demonstration of HYDRA and A3 repairing the previous CVE-2013-4854. The new vulnerability had no synthetic component, and it was easy to generate a malicious input. HYDRA repaired the vulnerability with a conditional disable, preventing any use of a request with the "tkey" additional record data. The final developer-generated patch had a more refined condition, but HYDRA's repair automatically fixed the live BIND server, and since none of the clients in our deployed environment used the tkey record, the fix allowed the DNS server to resume service immediately with no loss of operational capability.

6 Related Work

The A3 environment in which we implemented HYDRA is one of many [4][8][9][10][12] software resiliency and repair technologies.

If we consider code repair in response to an observed attack, ClearView (including Daikon [30], Heap Guard and Determina Memory Firewall) [2] is the most similar prior work to the combination of A3 and HYDRA. Like HYDRA, ClearView repairs deployed applications in response to invariant violations. The key differences are that HYDRA operates on either running binaries (via VMI policies) or on source code, and that HYDRA runs experiments within the A3 environment (in its experiment area) to test patches before deploying any repaired binaries. ClearView instead coordinates a set of deployed binaries to perform testing on live, in-use copies. A3 and Daikon also reason about invariants at different levels of abstraction. A3 focuses on application-level invariants, whereas Daikon considers in-memory data structures or register values. We believe that the A3+ HYDRA approach to repair is more likely to deploy working patches because it explicitly tests functionality. The focus on higher-level also helps the manual vulnerability mitigation process by providing a human-readable patch, representative test cases, and a chain of reasoning.

Work on *automated software repair* mostly follows a *generate-and-validate* approach [3] that relies on functional or unit tests to detect the presence of bugs and to assess the validity of discovered patches. Examples include GenProg [1], RSRepair [5], and AE [6]. These systems assume that the code necessary for repair already exists within the program source [7] and that a patch can be *evolved* from these ingredients. These systems also attempt to produce user-acceptable patches that do not require further refinement. HYDRA, in contrast, seeks to protect a given deployment and aid human analysis. As such, HYDRA's scope of possible repairs is narrower. It is also difficult to evaluate HYDRA's performance relative to these systems because Qi et al. [3] documented deficiencies in the patch evaluation infrastructure used by these systems. AE performs best among the three, with 3 out of 54 bugs successfully patched.

Qi et al. also introduce Kali in [3]. HYDRA is similar to Kali in that it exhaustively searches a tractable solution space and patches entail functionality reduction. Unlike Kali, HYDRA does not consider patches to code executed in both malicious and benign cases. HYDRA will only produce repair hypotheses concerning values or statements that are *always different* on the malicious traces. This narrows HYDRA's solution space even further than Kali's, though HYDRA does spend search time executing test panels that allow it to develop and refine repair hypotheses. Qi et al. use Kali as an example of how a fast and simple approach to software repair can be at least as effective as other approaches that may require more scrutiny. They explicitly do not advocate its use for real repairs. In contrast, HYDRA is simpler relative to many prior approaches, but addresses practical issues that are out of scope for Kali by creating its own tests from recorded traffic, preserving application functionality by avoiding code paths used by benign traffic, and producing human readable patch summaries.

Other approaches to automated repair include learning from existing human patches [14][16] and transferring correct patch code from *donor* applications to recipients [15]. HYDRA composes patches from explicit repair templates as opposed to drawing from an existing corpus of deployed code. Whether one approach is more generalizable than the other is unclear. Explicit repair templates

introduce less uncertainty in the modifications made to the application, but suffers from a narrower range of possible repairs. The observation that while attacks may have multiple manifestations, the underlying root causes are usually from a small set of programming errors motivated HYDRA's template approach.

Triage [13] attempts to debug and diagnose a deployed system by using replay for fault localization, root cause analysis, and testing repair hypotheses but does not implement automated code modification, testing, and deployment. FUZZBUSTER offers a capability more similar to A3's network filtering techniques than to HYDRA's code repair, and takes a proactive approach to program repair by fuzzing for vulnerabilities and automatically deriving regular expressions that block vulnerability triggering inputs [11].

7 Conclusion and Future Work

HYDRA automates a complex, expert-intensive portion of the post-incident response process. It localizes the vulnerable code exploited by a successful attack and provides code modifications that mitigate the attack. Implemented as a part of a larger *post-incident response workflow* enabled by managed execution environments like A3, HYDRA provides system administrators a quick (likely imperfect) fix, an analysis tool, and a development aid. Like most prior work [1][2][3][5], HYDRA's patches reduce the application functionality. HYDRA's diagnostics, repair candidates, and fitness scores of candidate repairs help reduce the overall patch time.

While initial results are promising, the area is ripe for more work. HYDRA applies a small set of test panels and simple repair templates, which allows exhaustive search of its relatively small solution space. One future avenue of exploration is to extend HYDRA by adding new test panels and repair templates that uses powerful capabilities like static analysis and taint tracking to reinforce the differential exploration of benign and malicious execution traces. Another much-needed direction is approaches to customize the search so that HYDRA can zero in on the vulnerable code and potential repair candidates faster when the set of test panels and repair templates are much larger. A final direction to explore is the usability of HYDRA as a human-in-the-loop *defense of last resort* for critical network-facing server applications, which will require user studies, field trials, and an automated, repeatable process for applying HYDRA to new application deployments.

ACKNOWLEDGMENTS

This work was supported by the US Air Force and the Defense Advanced Research Projects Agency (DARPA) under Contract No. FA8750-10-C-0242. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are of the authors and should not be interpreted as representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

REFERENCES

- [1] C. Le Goues, et al. "A systematic study of automated program repair: Fixing 55 out of 105 bugs for \$8 each." Software Engineering (ICSE), 2012 34th International Conference on. IEEE, 2012.

- [2] J.H. Perkins, et al. "Automatically patching errors in deployed software." Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. ACM, 2009.
- [3] Z. Qi, et al. "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems." Tech. report. (2015).
- [4] M. Kling, et al. "Bolt: on-demand infinite loop escape in unmodified binaries." ACM SIGPLAN Notices. Vol. 47. No. 10. ACM, 2012.
- [5] Y. Qi, et al. "The strength of random search on automated program repair." Proceedings of the 36th International Conference on Software Engineering. ACM, 2014.
- [6] W. Weimer, Z.P. Fry, and S. Forrest. "Leveraging program equivalence for adaptive program repair: Models and first results." Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on. IEEE, 2013.
- [7] M. Martinez, W. Weimer, and M. Monperrus. "Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches." Companion Proceedings of the 36th International Conference on Software Engineering. ACM, 2014.
- [8] M.C. Rinard, et al. "Enhancing Server Availability and Security Through Failure-Oblivious Computing." OSDI. Vol. 4. 2004.
- [9] P. Larsen, S. Brunthaler, and M. Franz. "Automatic Software Diversity." IEEE Security & Privacy 2 (2015): 30-37.
- [10] T. Jackson, et al. "Compiler-generated software diversity." Moving Target Defense. Springer New York, 2011. 77-98.
- [11] D.J. Musliner, et al. "Fuzzbuster: A system for self-adaptive immunity from cyber threats." Eighth International Conference on Autonomic and Autonomous Systems (ICAS-12). 2012.
- [12] S.J. Crane, et al. "It's a TRaP: Table Randomization and Protection against Function-Reuse Attacks." Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. ACM, 2015.
- [13] J. Tucek, et al. "Triage: diagnosing production run failures at the user site." ACM SIGOPS Operating Systems Review. Vol. 41. No. 6. ACM, 2007.
- [14] F. Long and M. Rinard. "Prophet: Automatic patch generation via learning from successful human patches." (2015).
- [15] S. Sidiropoulos-Douskos, et al. "Automatic error elimination by horizontal code transfer across multiple applications." (2015).
- [16] D. Kim, et al. "Automatic patch generation learned from human-written patches." Proceedings of the 2013 International Conference on Software Engineering. IEEE Press, 2013.
- [17] Symantec. ISTR20: Internet security threat report.
- [18] Akamai. Q3 2015 State of the internet – security report.
- [19] National Vulnerability Database. CVE2012-1823 summary. <<https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-1823/>>, Last visited Dec. 2017
- [20] National Vulnerability Database. "Vulnerability summary for CVE-2014-6271." <<https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-6271/>>, Last visited Dec. 2017
- [21] P. Pal et al. Advanced Adaptive Application (A3) environment: initial experience. In Proc. Middleware. ACM, 2011.
- [22] P. Pal, et al. A3: An environment for self-adaptive diagnosis and immunization of novel attacks. In Proc. IEEE Intl. Conf. on Self-Adaptive and Self-Organizing Systems Workshops. IEEE, 2012.
- [23] A. Paulos et al. Isolation of malicious external inputs in a security focused adaptive execution environment. Availability, Reliability and Security (ARES), 10th International Conference on. IEEE, 2013.
- [24] <<http://git-public.flux.utah.edu/gitweb.cgi?p=a3/vmi.git;a=summary/>>, a3/vmi.git/summary. Last visited Dec. 2017.
- [25] National Vulnerability Database. "Vulnerability summary for CVE-2014-0160." <<https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-0160/>>, Last visited Jan. 2018
- [26] The Xen Project. <<http://xenproject.org/>>, Loaded Dec. 2015.
- [27] S. Clark, et al. "Empirical evaluation of the A3 environment: evaluating defenses against zero-day attacks." Availability, Reliability and Security (ARES), 10th International Conference on. IEEE, 2015
- [28] MITLL CTF. <<https://events.ll.mit.edu/mitllctf/>>, Loaded Jan. 2016.
- [29] D. Johnson, M. Hibler, and E. Eide. "Composable multi-level debugging with Stackdb." ACM SIGPLAN Notices. Vol. 49. No. 7. ACM, 2014.
- [30] M.D. Ernst, et al. "The Daikon system for dynamic detection of likely invariants." Science of Computer Programming 69.1 2007.
- [31] L. Bilge and T. Dumitras. Before we knew it: an empirical study of zero-day attacks in the real world. In Proc. ACM Conference on Computer and Communications Security, Oct. 2012.
- [32] SANS Institute, Protection from the Inside: Application Security Methodologies Compared, <<https://www.sans.org/reading-room/whitepapers/analyst/protection-inside-application-security-methodologies-compared-35917/>>, Last visited Dec 2017