Online Network Forensics for Automatic Repair Validation

Michael E. Locasto[1], Matthew Burnside[2], and Angelos D. Keromytis[2]

[1] Institute for Security Technology Studies, Dartmouth College
[2] Department of Computer Science, Columbia University

Abstract. Automated intrusion prevention and self-healing software are active areas of security systems research. A major hurdle for the widespread deployment of these systems is that many system administrators lack confidence in the quality of the generated fixes. Thus, a key requirement for future self-healing software is that each automatically-generated fix must be validated before deployment. Under the response rates required by self-healing systems, we believe such verification must proceed automatically. We call this process Automatic Repair Validation (ARV). We describe the design and implementation of *Bloodhound*, a system that tags and tracks information between the kernel and the application and correlates symptoms of exploits (such as memory errors) with high-level data (*e.g.*, network flows). By doing so, Bloodhound can replay the flows that triggered the repair process against the newly healed application to help show that the repair is accurate (*i.e.*, it defeats the exploit). We show through experimentation a performance impact of as little as 2.6%.

keywords: software self-healing, automatic repair validation

1 Introduction

Recent advances in self-healing software techniques have paved the way for autonomic intrusion reaction, but real-world deployments of such systems have lagged behind research efforts. The limits of detection technology have historically mandated that researchers address the shortcomings of intrusion detection before reaction mechanisms (*i.e.*, self-healing mechanisms) are considered – an attack must be detected before a response can be mounted. In addition, many system administrators are reluctant to allow a defense system to make unsupervised changes to the computing environment, even though (and precisely because) a machine can react much faster than a human.

1.1 Automatic Repair Validation

Automatically generated fixes must be subjected to rigorous testing in an automated fashion. This problem is the essence of Automatic Repair Validation (ARV), a new area of intrusion defense research. ARV encompasses the entire spectrum of an automated response system's functionality: attack detection, repair accuracy, repair precision, and impact on normal behavior:

1. **Validation of detection** – The system must verify that the events causing an alert actually produce a compromise. In the case where the sensor is an anomaly detector, the detector's initial classification must be confirmed.
2. **Validation of a repair's accuracy** – The system must test and verify that the repair defeats at least the exploit input that triggered the detection. The rest of the paper discusses the challenges of this process and our solution to it in detail.
3. **Validation of a repair's precision** – The fix must be precise, in the sense that it blocks malicious variants of the original attack. For example, if the fix is an input filter, the system must ensure that the signature generation does not fall prey to the allergy attack [1], whereby a signature generation system is trained to correlate benign input with undesireable symptoms, thereby increasing the rate of false positives.
4. **Validation of a repair's impact on application behavior** – Behavior exhibited by the application after self-healing should be similar to the previous behavior profile of the application. Section 6 briefly discusses an approach for addressing this challenge.

Even though the other problems in this space remain important challenges, we concern ourselves with a repair's accuracy in this paper. Verifying accuracy requires the identification and replay of the attack inputs. Identifying these inputs is challenging, as they may not have been captured correctly (or at all) by the defense instrumentation. The challenge is greater if the input is contained in network traffic — data that most humans find difficult to rapidly analyze by hand. Although the general ARV problem exists for many types of systems, we propose a solution for systems that deal with network traffic. These applications remain popular targets for attack due to the relative anonymity of an attacker and the ease with which input can be sent to the system.

1.2 Correlating Across Abstraction

ARV reveals the tension between the need for abstraction during system design with the need for anti-abstraction during system healing. Since defense system components may not operate at the same layer, the underlying challenge in this problem space is correlating information across layers of abstraction. For example, a detection component may perform binary supervision, but its defense may be the generation of signatures that match network packet content. The accuracy of such signatures is suspect because the data that trips the detector is no longer contained in IP packets; rather, it resides in a memory address and may have been repeatedly transformed before detection. Without appropriate instrumentation, the detector has lost the details of how the data arrived in the memory location. This paper presents a system that contains instrumentation to address this problem.

Abstraction is a powerful tool for system design. The key assumption is that a component should be self-contained: it should not know (and therefore have a hidden dependency on) the implementation details of another component. This assumption greatly eases design cost and increases the flexibility of the system's composition. It no longer

holds when a system must self-heal. Other components must be aware of the details of the malfunctioning component so that they can respond to the failure[3].

Self-healing systems must correlate events and data across layers of abstraction. This correlation should remain flexible: if a component switches to a different implementation, the self-healing infrastructure should adapt to the internal behavior of the new component. We plan to identify a core set of anti-abstraction design patterns in follow-on work to help support this capability.

This paper considers the design, architecture, and implementation of *Bloodhound*, a system for recording and replaying attack input embedded in network flows. The primary challenge involves correlating the form of input that triggers the self-healing instrumentation with the input that originally enters the system. As such, Bloodhound correlates network flows with host-based events by marking an application's internal data structures with the ID of the flow that "taints" that data structure. Bloodhound breaks the traditional abstraction between low-level network data and high-level application data objects by placing sensors at multiple layers in the network stack and application stack. Specifically, Bloodhound runs on the OpenBSD operating system, and it uses a loadable kernel module for a pseudo-device to break the user/kernel-space abstraction.

2 Related Work

ARV is only meaningful if a system protects itself with a self-healing mechanism. Rinard *et al.* [2] have developed compiler extensions that insert code to deal with access to unallocated memory. This technique is leveraged for *failure-oblivious computing*. A related idea is that of *error virtualization* [3], which creates a mapping between the set of errors that could occur during a program's execution and the limited set of errors that are explicitly handled by the program code. The Rx system [4] improves on these approaches by performing only safe perturbations of application state to execute through a fault.

2.1 Signature Generation

Although we could leverage Bloodhound to generate exploit signatures, such a task is not our goal. In addition, recent work [5, 6] has called into question the ultimate utility of exploit-based signatures, and Cui *et al.* [7] discuss using binary-level taint-tracking to construct network or filesystem level "data patches" to filter input instances related to a particular vulnerability. Newsome *et al.* suggest generating and distributing vulnerability-specific execution filters [8] based on the identification of a particular control flow path derived from tainted dataflow analysis. Many systems aim at automatically generating signatures of malicious traffic [9–12]. To generate a signature, most of these systems either examine the content or characteristics of network traffic or instrument the host to identify malicious input.

[3] Note that the problem domain we consider in relation to "self-healing" is *not* the more traditional fault-tolerant environment for distributed applications, where transparent fail-over to replicated components is the norm. In such cases, anti-abstraction serves little purpose.

Other recent work takes a hybrid approach and performs host-type processing on network data. Abstract Payload Execution (APE) [13] identifies network traffic that contains malicious code by treating the content of a packet as machine instructions. Instruction decoding of packets can identify the sequence of instructions in an exploit whose purpose is to guide the program counter to the exploit code. Kruegel *et al.* [14] detect polymorphic worms by learning a control flow graph for the worm binary with similar techniques. *Convergent static analysis* [15] also aims at revealing the control flow of malcode.

DIRA [16] is a compiler extension that adds instrumentation to keep track of memory operations and check the integrity of control flow transfers. It creates a string-based signature for filtering further exploit instances. Liang and Sekar [12] and Xu *et al.* [17] concurrently proposed using address space randomization to drive the detection of memory corruption vulnerabilities and create a signature to block further exploits. These systems operate in a similar fashion to Bloodhound in that they trace back from a memory error to network data. The work of King and Chen [18] utilizes virtual machine logging and replay to step back through checkpoints of a system to identify the ultimate source of a compromise.

Newsome *et al.* propose dynamic taint analysis [19] to detect exploited vulnerabilities. The Vigilante [20] system uses similar analysis for detection and defines an architecture for production and verification of Self-Certifying Alerts (SCAs), a data structure for exchanging information about newly discovered vulnerabilities. The verification step is an example of the form of ARV mentioned in Section 1: Vigilante verifies the control flow path that forms the basis for the alert actually causes an exploit to occur. While Bloodhound uses tainted dataflow analysis, it is not a replacement for such systems. Instead, it augments dataflow analysis systems by considering how tainted data flows through the kernel as well as userspace memory.

2.2 Replaying Traffic

Bloodhound archives all flows consumed by an application and replays only those flows that were related to the exploit in question. Replaying application protocol dialogs is a crucial aspect of an ARV system, and it proves useful in a number of situations (*e.g.*, application or protocol debugging). Traffic can be reproduced in two major ways. First, the raw packets can be recorded and replayed, but this approach may require a large amount of storage and further packet processing. The second approach builds an analytical model of traffic and then generates traffic matching these characteristics.

TCPopera [21] can interactively replay network traffic. It is broadly applicable to problems that require producing large amounts of realistic network data. Roleplayer by Cui *et al.* [22] and the ScriptGen system by Leita *et al.* [23] attempt to reconstruct and replay application-level messages from network flows with little contextual data and a few guiding heuristics.

The Replayer system [24] formalizes the problem of application replay. Replayer describes a sound approach (that is, one not based on heuristics) to generate and issue an input that directs Host Bob to reach the same state as Host Alice (as determined by some post-condition test). Although our replay problem is somewhat different, the Replayer system is the most closely related research effort to Bloodhound, and we found the

syntax of Replayer's formal model of application protocol replay useful to help frame Bloodhound's task.

The most significant difference between Bloodhound and Replayer is that Bloodhound focuses on identifying whether a particular network flow in the database of stored flows contains malicious input. In addition, the underlying problem differs; Replayer is designed to suitably transform a traffic trace or input so that a second host reaches a state equivalent to the first host. Replayer does this by, for example, changing parts of the input to reflect the appropriate hostname or cookie value based on the post-condition constraint. In contrast, Bloodhound leverages taint analysis to identify the set of network packets involved in an exploit. We designed Bloodhound to work with a self-healing system. Since the host is modified via self-healing, the target state explicitly differs from the initial observed state (*i.e.*, the application reaches a new "healed" state rather than the same corrupted state).

One interesting avenue of research would be to combine Replayer and Bloodhound so that stored flows that Bloodhound has selected for replay are consistent with the state of the application and external world. The major challenge with this approach is to ensure that fields critical to the exploit are not changed in a way that interferes with the exploit's efficacy, thus introducing a false negative into the validation process (*i.e.*, the systems believes the exploit was defeated by the self-healing when it was simply broken by the replay engine).

3 Design Space

ARV consists of automatically validating each step in the process toward a repair. We focus our discussion here on how to determine the accuracy of a repair for network applications. Bloodhound's tasks include (a) preferentially recording network traffic, (b) searching through these flows after the application has been healed, and (c) replaying the relevant flows to test this repair. Bloodhound provides evidence that an automated response protects against the input that triggered the self-healing mechanism.

3.1 ARV Replay Definition

ARV replay involves selecting a series of packets to transmit to a modified version of an application to test whether it survives the act of consuming those packets. An ARV replay system, in effect, reprises the role of the attacker. Consider a program P and a set of network flows F. Some subset of those network flows are exploit flows e.

$$\{e_0, e_1, \ldots, e_n\} \subseteq F \tag{1}$$

When P runs and receives input F containing exploit flows, it enters an exploited or error state σ (according to some detection mechanism).

$$\text{Run}(P, F) \rightarrow \sigma \tag{2}$$

After P consumes F and reaches the exploited or error state σ, a self-healing function H operates on P and σ to produce a "healed" program P', optionally replacing σ or other states with correct or healed versions according to the repair strategy in use.

$$H(P, \sigma) \rightarrow P' \tag{3}$$

The ARV accuracy test is to identify the subset $e_0, e_1, \ldots, e_n$ of F and verify that:

$$\text{Run}(P', \{e_0, e_1, \ldots, e_n\}) \nrightarrow \sigma \tag{4}$$

that is, to determine whether the healed version of the application enters an error or exploit state on replay of the attack traffic.

During normal operation of P, (*i.e.,* when it has not entered σ), the system records a database of flows F where each flow f_i consists of incoming and outgoing packets. Each packet of f_i causes changes in user and kernel memory and expresses a particular control flow path in P. Instrumenting memory accesses to observe changes derived from each f_i results in a directed graph D. Each node in D is a memory address, and each edge in D indicates that the source node "taints" the destination node. We label each edge with the instruction responsible for propagating the taint.

While creating D seems fairly straightforward, D — as described — does not contain enough information to support precise traceback. Traceback becomes imprecise when the OS or application reuses a memory location to hold data derived from both malicious and non-malicious flows. At such "common point" memory locations (like a buffer that stores incoming requests), D does not prevent the traceback routine from mistakenly expanding its scope. Common points have a fan-in from multiple flows that becomes a fan-out during traceback.

Table 1. *Forward-Marking Supports Precise Flow Traceback.* The routine adds a node to D, labels the transition, and labels the new (or existing) target node. Traceback iterates over E to collect the malicious flow subsets from the corresponding nodes of D.

```
ROUTINE PROPAGATE(MEM SRC, MEM DST, FLOW F)
    create new edge in D from src to dst
    addFlow(dst, f)

ROUTINE TRACEBACK(EXPLOITMEM E, TAINTGRAPH D)
    foreach memory address x in E
        foreach node y in D
            if x = y
                maliciousFlows ← getFlows(y)
    return maliciousFlows
```

Our solution adopts a simplified traceback approach based on a *forward marking* scheme that labels each node (memory address) in D with the flow ID responsible for the current change in the node's data. Once P enters σ, the detector generates a set of memory addresses E that are involved in the exploit (*e.g.,* an address whose

contents enters the instruction register). Forward marking continuously maintains the information necessary to quickly derive $\{e_0, e_1, \ldots, e_n\}$ (the flows responsible for the exploit) given E. At that point, Bloodhound performs the accuracy test of Equation 4 to determine if P' represents a viable candidate to replace P in production service.

3.2 Traffic Recording

Many options exist for preferentially recording flows. The simplest approach archives all network traffic and replays the entire archive on demand. This approach seems untenable; a system can only store a finite volume of traffic, and replaying or searching an arbitrarily large archive is infeasible in a timing-dependent domain like self-healing software. The flow archive's size must result in a reasonable duration for searching and replaying flows. We consider some heuristics for choosing which flows to store:

- *Save a sliding window of the last* n *days worth of traffic.* This heuristic is simple to implement, and it is simple to tune the size of the archive to optimize storage or replay duration. To test a patch against the archive, simply replay all stored flows. The obvious downside to this heuristic is that it fails against attacks that last longer than, or started before, the n days stored in the archive.
- *Save a probabilistic window.* Rather than using a window with a fixed horizon, probabilistically eject flows from the archive as they age. Aging policies can range from FIFO to LRU to random ejection. Regardless of the aging policy chosen, the self-healing software can only make probabilistic statements about its confidence in a particular patch, based on the likelihood that the entire exploit was contained in the archive.
- *Archive flows flagged by a signature-based misuse detector (e.g., Snort).* This heuristic has the advantage of simplicity, but it is a poor match for a self-healing system that can patch against new or `0`-day attacks. The self-healing system can patch against never-before-seen attacks, but the testing framework would could only replay flows for which a signature already exists.
- *Archive those flows identified by a payload-based anomaly detection system such as Anagram [25].* This heuristic supports the detection of suspect flows that have never been seen before, unlike the previous example. The viability of this heuristic depends entirely on the abilities of the anomaly detection system. If the anomaly detection system has a low false negative rate, then the likelihood that the entire exploit package is archived is very high.
- *Archive flows based on tainted dataflow analysis.* This heuristic focuses on reducing the duration of testing, rather than reducing storage space. As an application and the OS handle each flow, the flow will taint various user and kernel data structures. If each flow is indexed based on the data structures it taints, then during testing only those flows which taint the data structures involved in the patch under question require replay.

These heuristics may also be combined to achieve various points on the tradeoff curve between archive size, replay duration, and confidence. While the choices of recording strategies listed above all involve tradeoffs, we choose to employ a solution

that does not admit false negatives during detection. That is, during replay, we prefer to replay flows we *know* to be malicious. Thus, Bloodhound uses a form of host-based taint analysis to help identify flows that are involved in a particular exploit.

3.3 Classes of Attack

We classify attacks into several broad categories to make it easier to illustrate the potential complexity of the recording and replay task. The simplest scenario occurs when a single TCP flow or UDP packet encompasses the entire attack (*e.g.*, worms like Slammer or Code Red). The attacker initiates a connection, transmits the exploit code, and then closes the connection. Testing must replay the entire flow. In a slightly more complex version, the attack may be a subset of a larger, innocent flow. Consider an SSH connection where the attacker behaves innocently for several hours and then runs an exploit. We differentiate this attack class from the previous scenario because it may not be desirable to store all of a long-duration flow if only a portion is suspicious.

An attack can be distributed across multiple TCP flows, each of which taken alone appears innocent. Consider a hypothetical attack where one flow corrupts a buffer and a second one delivers the rest of the exploit. To complicate matters, an attack may take advantage of the timing relationship between multiple flows or the timing relationship between packets of a single flow. Attacks of this form exploit race conditions in multi-threaded code, and recreating the circumstances of race conditions is notoriously difficult. At the very least, the timing relationship between the packets or flows must be preserved for testing. This requirement presents difficulties for a *rapid* automatic response, as deployment time is constrained by characteristics of the attack – parameters that the attacker controls. An attacker can potentially distribute exploits from both these scenarios over arbitrarily many flows.

The attack may depend on innocent user action. Consider an exploit where the adversary sends an attack packet, followed by N innocent requests, the combination of which triggers the exploit. Any replay of the flows must include the attack packet *and* all N innocent requests. This class is particularly difficult because, to the untrained eye, the N^{th} packet is highly suspect, while the true attack packet does not necessarily stand out. Worst of all, regardless of the validity of a particular patch, testing against the N^{th} packet almost always succeeds because it is an innocent request.

Finally, an attack may be polymorphic. That is, the algorithm for generating the exploit code may use cryptographic or other heuristics to change the form of the attack over time. As a result, searching a flow or flows for particular bit patterns cannot necessarily identify attacks of this form. In the most pathological case, an attack may consist of any combination of the above attack classes: an attack may be polymorphic, spread across multiple TCP flows, *and* depend on innocent user action. While we do not expect this situation to dominate, it is useful to illustrate the extent of the design space.

4 Bloodhound Implementation

Based on the constraints we present in Section 3, we cover the implementation of Bloodhound's major components, including the structure of our data storage component as

well as the implementation of a pseudo-device and its communication protocol with the user space data management framework. We discuss ways to optimize traffic recording through the use of a network content anomaly detector in Section 6.

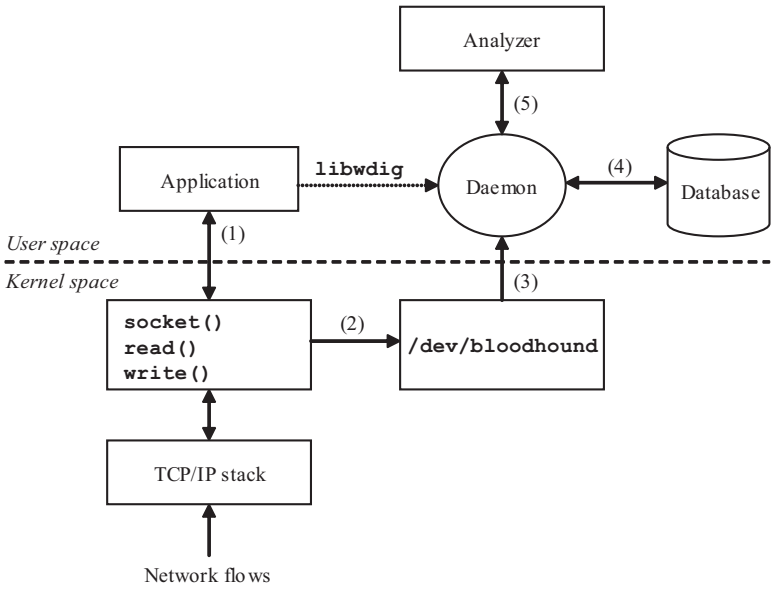

Fig. 1. *Bloodhound Architecture*. In order to track network flows from the operating system to user applications, Bloodhound must break the user space/kernel abstraction. When an application issues a network I/O system call (1), Bloodhound copies the network flow information to a pseudo-device `/dev/bloodhound` (2). A daemon (3) regularly polls the pseudo-device and then stores the network flow data in a database (4). After a self-healing repair has been effected, the analyzer can issue queries to the database (5) to discover which flow caused the exploit. The flow can then be replayed against the newly healed application to test the efficacy of the repair.

4.1 Architecture & Operation

Bloodhound must store flow information and enough structured forensic information to link flows with kernel and application data items, so Bloodhound's core components are distributed across the kernel and user space as shown in Figure 1.

The `/dev/bloodhound` pseudo-device is implemented as an OpenBSD Loadable Kernel Module. The pseudo-device supports an `ioctl()` call `BH_PID` which indicates the process ID number of specific process to observe. On loading, the module hooks into the I/O system calls to intercept network data before it is copied to user space. Each flow associated with the observed process is copied to an in-kernel buffer which is read through standard `read()` calls on `/dev/bloodhound`.

Data is read out of `/dev/bloodhound` by a simple daemon process which uses a Berkeley DB 4.5 database to store flows and related information. The database maintains two primary relationships: a mapping from flow IDs to flow objects (a data struture that we define), and a mapping from tainted memory addresses to flow IDs. We created a library (`libwdig`) to provide access to the data store and define our "flow" data type. The library contains the following core procedures:

- **Register Flow** – This procedure creates a message that encodes a flow and sends it via the protocol described below to the daemon. The daemon parses the message and inserts the flow into the database indexed by the flow ID.
- **Register Binding** – This procedure creates a message that contains a memory address and a flow ID. The daemon inserts this mapping into the database.
- **Retrieve Flow** – This procedure provides a physical memory address to the daemon. If the address points to a flow, the daemon returns the associated flow ID.
- **Retrieve Location** – This procedure supplies the daemon with a flow ID. As a result, the daemon traverses the bindings of memory addresses to flow IDs and returns a list of memory addresses that are associated with the given flow ID.

The user space daemon employs `libwdig` to mediate an application's access to the database. User applications can also call `libwdig`'s functions directly to "manually" store trace information. We plan to investigate methods of automatically performing this dataflow analysis using binary rewriting to inject calls to `libwdig` functions.

The daemon serves as the hub of the system. It manages communication with three different components. First, it intercepts any user space applications that use the `libwdig` library directly. The daemon receives requests for flows or memory binding insertions into the database and invokes the appropriate database functions to handle the request. Second, the daemon communicates with the kernel through the `bloodhound` device. The daemon periodically polls the bloodhound device for messages. The daemon retrieves the messages and checks to see whether the flow IDs associated with the messages have been entered in the database. If they have not, it registers new flows in the database with the new flow IDs. Finally, the daemon helps the Analyzer perform forensic operations by receiving requests for memory address or flow ID lookups and passing them to the database. The daemon uses a TCP-based protocol to communicate with the Analyzer, as described in Table 2.

5 Evaluation

We evaluate Bloodhound by examining three aspects of the system. First, we discover what impact Bloodhound has on the normal, *i.e.,* non-repair-time, performance of an application by characterizing the slowdown due to system call interposition of network I/O related calls. Second, we employ an end-to-end test of the system to show how Bloodhound's forensic capability works for a synthetic vulnerability. Finally, we consider the utility of a possible optimization that employs either signature or statistical content anomaly approaches to pre-classify (and thus limit) the number of flows that Bloodhound would have to store for any given process.

Table 2. *Message Structure for Daemon Communication.* A message consists of two control bytes: the first indicates either query `Q` or response `R`, and the second distinguishes between a memory location `L`, a flow `F`, or a flow ID `I`. The rest of the message encodes a serialized representation of the memory location or flow.

Direction	Message Format	Meaning
To daemon	`Q, I, flow_id, flow`	Register `flow`
	`Q, L, loc, flow_id`	Register a binding
	`Q, L, loc`	Retrieve `flow_id` at location `loc`
	`Q, I, flow_id`	Retrieve `flow_id`
To client	`R, F, flow`	Response to retrieve `flow`
	`R, I, flow_id`	Response to register `flow ID`

5.1 Performance

To understand the performance impact of Bloodhound, we instructed the `bloodhound` device to observe a single-process instance of the Apache web server. The web server was running on a Dell PowerEdge 2650 with a 2.8GHz Intel Xeon processor and 1GB of RAM. We used a second, identical, Dell PowerEdge 2650, connected to the first over Gigabit Ethernet to download a collection of files. The collection of files we chose was the Apache web server manual, as distributed with the OpenBSD operating system. It consists of 163 text and graphics files totalling 2.1M of data. The manual was downloaded 25 times with `/dev/bloodhound` inactive, followed by another 25 downloads with `/dev/bloodhound` activated. On average a download with `/dev/bloodhound` inactivate took 4.19s, while a download with `/dev/bloodhound` active took 4.31s, for a general performance impact of 2.6%.

5.2 Efficacy

The purpose of our end-to-end efficacy test is to illustrate how Bloodhound can work back from a memory error or other indication of a detected attack to the network flow that contained the exploit input. Memory errors manifested in signals like SIGSEGV are common symptoms of detected exploits, especially when protection mechanisms like Stackguard, address space layout randomizationm or instruction set randomization are employed. In any case, Bloodhound assumes that such a signal will be raised, and that the Analyzer can obtain a memory address to start working backward from. In future work, we plan to use a binary rewriting tool to track tainted data between application-level data structures.

We test the traceback process for a synthetic vulnerability. Our hypothesis is that we can use Bloodhound's kernel instrumentation to track an attack back from a memory error to the flow causing the error. Our experiment is based on a function containing a stack-based buffer overflow in an echo server. The echo server reads user input into a small buffer and echos it back to the user. An exploit script sends a long string of input characters to the server, causing an overwrite of the stack. Each time the `read` system call is used, it uses `/dev/bloodhound` to notify the database of the contents boundaries of the data transfer.

When an overflow occurs and the function returns, the OpenBSD stack-smash protection is triggered and causes the program to crash, dumping core. A script loads the core and determines the location of the RET value on the stack. This value is used as the search value in the database, identifying the dataset that triggered the overflow.

6 Discussion

Automatically validating a repair is a rich field for future work, especially for techniques to ensure that the behavior of an application after repair matches a profile of behavior known to be "good" or clean of malicious influence. A number of considerations exist in this space that parallel the challenges that Bloodhound faces. In particular, the choice of behavior aspects to record and analyze is a key to balancing the tradeoff between the amount of information retained and the ability to confirm with enough specificity that the system can automatically distinguish between known-good tested behaviors and anomalous or malicious behaviors. We believe that a promising approach would start with the rich set of techniques previously proposed for system call anomaly detection [26–28]. Capturing aspects of both data and control flow [29], including library, application, and system calls as well as function return values and arguments [30] seems like it would provide a solid profile with enough information to distinguish between these behaviors.

6.1 Limitations

Bloodhound's implementation can be improved along three lines. First, we do not deal with taint-tracking through the application itself. Such taint-tracking can be accomplished by a programmer making direct calls into our taint-tracking library. We can also explore the combination of our kernel-level taint-tracking with existing binary-level tainted dataflow analysis. Second, we plan to incorporate behavior profiling (as described above) into Bloodhound so that it can verify an application's post-healing profile. We are currently porting our OpenBSD implementation to Linux to support these capabilities.

A third area of future work deals with improving Bloodhound's playback capabilities to handle some of the more advanced classes of attack we list in Section 3.3. For example, while other "innocent" user packets may have set up the attack (*e.g.*, by causing some limit to be exceeded), Bloodhound does not necessarily identify them as involved in the exploit. We defer research on these types of attacks; the goal of our current research and development is to provide an infrastructure — currently absent — for addressing them. We believe, however, that repair validation can proceed in the presence of these types of attacks, and we offer a sketch of one possible way forward: the use of continuous hypothesis testing that proceeds through each level of attack. If the cost of applying the fix to a production system can be kept relatively low, Bloodhound can validate the repair in stages, where each stage assumes that the attack was more sophisiticated and constructs appropriate playback scenarios as needed. This incremental process of generating and validating fixes allows Bloodhound to both quickly validate a fix for a specific version of the exploit and continuously ensure that the fix

works against more advanced attack scenarios. The exploration of this type of intrusion defense system seems valuable, and Bloodhound's provides a framework so that researchers can implement more intelligent playback and testing strategies.

One potential criticism of Bloodhound is that it appears to be exploit-specific and therefore does not provide protection that vulnerability-specific systems might. While Bloodhound focuses on identifying a particular exploit input, its task does not conflict with the goals of vulnerability-specific defense systems [6, 8, 20, 7] and related analysis [5, 31]. Instead, Bloodhound can provide these systems with more confidence that the fix is correct and blocks *at the very least* the malicious input that triggered the instrumentation. Future work can use the input traffic as a template to generate other semantically correct instances of the flow so that the fix can be tested against a variety of inputs that exploit the same vulnerability. Systems similar to RolePlayer [22] or Replayer [24] seem well suited to this task. Cui *et al.* [7] illustrate the process of deriving practical signatures of previously unknown vulnerabilities. These "data patches" are generated in part by binary-level taint-tracking and help filter related exploit inputs.

6.2 Further Optimizations

We have performed several preliminary experiments to determine the feasibility of reducing the amount of traffic that ARV systems like Bloodhound need to store. ARV is essentially a problem of search; performing an online search, even of an indexed corpus, can be sped up if the size of the corpus is reduced. A system that only considers packets relevant to the current vulnerability or exploit would prove useful. Bloodhound uses taint propagation to achieve this measure. However, an ARV system can complement this dataflow analysis with packet classification schemes, including payload anomaly detection. We refer the interested reader to our technical report, which has the details [32].

7 Conclusions

Most attacks occur rapidly enough to frustrate manual defense or repair. It appears that defense systems must include some degree of autonomy. Recent advances have led to an emerging interest in self-healing software as a solution to this problem. System owners, however, are understandably reluctant to permit automated changes to their environment and applications in response to attacks. Testing an automatic repair helps raise the confidence level in self-healing systems. One critical part of such testing is the verification that the changes made by the self-healing mechanism actually defeat the original attack or close variations thereof.

This paper identifies the important challenge of Automatic Repair Validation (ARV): using audit information to test the resilience and efficacy of a self-healing repair. We present *Bloodhound*, a system for recording and replaying network flows related to the exercise of a particular vulnerability. The design process reveals a number of challenging problems that the research community needs to address in order to make self-securing systems a reality. Our implementation and experiments illustrate that the problem is surmountable; the performance impact on normal operation due to monitoring

seems reasonable, and the system can trace back to the flow at fault. In the future, we plan to deploy Bloodhound in a testbed like DETER to test how well Bloodhound can provide an audit service to other nodes.

Acknowledgments

We encountered the problem of automated verification of repairs when we built a system capable of self-healing network server applications against buffer overflow attacks. We would like to recognize the hard work of the other developers, Stelios Sidiroglou and Gabriela Cretu. In addition, we would like to thank Felix Wu and Seung-Sun Hong for sharing TCPopera with us. Finally, Ke Wang provided us with Anagram and was instrumental in helping us to run it.

This paper reports on work that was supported in part by ARO/DHS contract DA W911NF-04-1-0442, USAF/AFRL contract FA9550-07-1-0527, and NSF Grant 06-27473, with additional support from Intel Corporation. The content of this work is the responsibility of the authors and should not be taken to represent the views or practices of the U.S. Government or its agencies.

References

1. Chung, S.P., Mok, A.K.: Allergy Attack Against Automatic Signature Generation. In: Proceedings of the 9^{th} International Symposium on Recent Advances in Intrusion Detection (RAID). (2006)
2. Rinard, M., Cadar, C., Dumitran, D., Roy, D., Leu, T., W Beebee, J.: Enhancing Server Availability and Security Through Failure-Oblivious Computing. In: Proceedings 6^{th} Symposium on Operating Systems Design and Implementation (OSDI). (December 2004)
3. Sidiroglou, S., Locasto, M.E., Boyd, S.W., Keromytis, A.D.: Building a Reactive Immune System for Software Services. In: Proceedings of the USENIX Annual Technical Conference. (April 2005) 149–161
4. Qin, F., Tucek, J., Sundaresan, J., Zhou, Y.: Rx: Treating Bugs as Allergies – A Safe Method to Survive Software Failures. In: Proceedings of the Symposium on Systems and Operating Systems Principles (SOSP). (2005)
5. Brumley, D., Newsome, J., Song, D., Wang, H., Jha, S.: Towards Automatic Generation of Vulnerability-Based Signatures. In: Proceedings of the IEEE Symposium on Security and Privacy. (2006)
6. Wang, H.J., Guo, C., Simon, D.R., Zugenmaier, A.: Shield: Vulnerability-Driven Network Filters for Preventing Known Vulnerability Exploits. In: Proceedings of the ACM SIGCOMM. (August 2004)
7. Cui, W., Peinado, M., Wang, H.J., Locasto, M.E.: ShieldGen: Automated Data Patch Generation for Unknown Vulnerabilities with Informed Probing. In: Proceedings of the IEEE Symposium on Security and Privacy. (May 2007)
8. Newsome, J., Brumley, D., Song, D.: Vulnerability–Specific Execution Filtering for Exploit Prevention on Commodity Software. In: Proceedings of the 13^{th} Symposium on Network and Distributed System Security (NDSS 2006). (February 2006)
9. Kim, H.A., Karp, B.: Autograph: Toward Automated, Distributed Worm Signature Detection. In: Proceedings of the USENIX Security Conference. (2004)

10. Singh, S., Estan, C., Varghese, G., Savage, S.: Automated Worm Fingerprinting. In: Proceedings of Symposium on Operating Systems Design and Implementation (OSDI). (2004)
11. Newsome, J., Karp, B., Song, D.: Polygraph: Automatically Generating Signatures for Polymorphic Worms. In: Proceedings of the IEEE Symposium on Security and Privacy. (May 2005)
12. Liang, Z., Sekar, R.: Fast and Automated Generation of Attack Signatures: A Basis for Building Self-Protecting Servers. In: Proceedings of the 12^{th} ACM Conference on Computer and Communications Security (CCS). (November 2005)
13. Toth, T., Kruegel, C.: Accurate Buffer Overflow Detection via Abstract Payload Execution. In: Proceedings of the 5^{th} International Symposium on Recent Advances in Intrusion Detection (RAID). (October 2002) 274–291
14. Kruegel, C., Kirda, E., Mutz, D., Robertson, W., Vigna, G.: Polymorphic Worm Detection Using Structural Information of Executables. In: Proceedings of the 8^{th} International Symposium on Recent Advances in Intrusion Detection (RAID). (September 2005) 207–226
15. Chinchani, R., Berg, E.V.D.: A Fast Static Analysis Approach to Detect Exploit Code Inside Network Flows. In: Proceedings of the 8^{th} International Symposium on Recent Advances in Intrusion Detection (RAID). (September 2005) 284–304
16. Smirnov, A., Chiueh, T.: DIRA: Automatic Detection, Identification, and Repair of Control-Hijacking Attacks. In: Proceedings of the 12^{th} Symposium on Network and Distributed System Security (NDSS). (February 2005)
17. Xu, J., Ning, P., Kil, C., Zhai, Y., Bookholt, C.: Automatic Diagnosis and Response to Memory Corruption Vulnerabilities. In: Proceedings of the 12^{th} ACM Conference on Computer and Communications Security (CCS). (November 2005)
18. King, S.T., Chen, P.M.: Backtracking Intrusions. In: 19^{th} ACM Symposium on Operating Systems Principles (SOSP). (October 2003)
19. Newsome, J., Song, D.: Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In: Proceedings of the 12^{th} Symposium on Network and Distributed System Security (NDSS). (February 2005)
20. Costa, M., Crowcroft, J., Castro, M., Rowstron, A.: Vigilante: End-to-End Containment of Internet Worms. In: Proceedings of the Symposium on Systems and Operating Systems Principles (SOSP). (2005)
21. Hong, S.S., Wu, S.F.: On Interactive Internet Traffic Replay. In: Proceedings of the 8^{th} International Symposium on Recent Advances in Intrusion Detection (RAID). (September 2005) 247–264
22. Cui, W., Paxson, V., Weaver, N.C., Katz, R.H.: Protocol-Independent Adatpive Replay of Application Dialog. In: Proceedings of the 13^{th} Symposium on Network and Distributed System Security (NDSS 2006). (February 2006)
23. Leita, C., Mermoud, K., Dacier, M.: ScriptGen: an automated script generation tool for honeyd. In: ACSA 2005, 21st Annual Computer Security Applications Conference, December 5-9, 2005, Tucson, USA. (Dec 2005)
24. Newsome, J., Brumley, D., Franklin, J., Song, D.: Replayer: Automatic Protocol Replay by Binary Analysis. In: Proceedings of the 13^{th} ACM Conference on Computer and Communications Security (CCS). (2006) 311–321
25. Wang, K., Parekh, J.J., Stolfo, S.J.: ANAGRAM: A Content Anomaly Detector Resistant To Mimicry Attack. In: Proceedings of the 9^{th} International Symposium on Recent Advances in Intrusion Detection (RAID). (2006)
26. Gao, D., Reiter, M.K., Song, D.: Gray-Box Extraction of Execution Graphs for Anomaly Detection. In: Proceedings of the ACM Conference on Computer and Communications Security (CCS). (2004)

27. Feng, H.H., Kolesnikov, O., Fogla, P., Lee, W., Gong, W.: Anomaly Detection Using Call Stack Information. In: Proceedings of the 2003 IEEE Symposium on Security and Privacy. (May 2003)
28. Giffin, J.T., Dagon, D., Jha, S., Lee, W., Miller, B.P.: Environment-Sensitive Intrusion Detection. In: Proceedings of the 8^{th} International Symposium on Recent Advances in Intrusion Detection (RAID). (September 2005)
29. Bhatkar, S., Chaturvedi, A., Sekar., R.: Improving Attack Detection in Host-Based IDS by Learning Properties of System Call Arguments. In: Proceedings of the IEEE Symposium on Security and Privacy. (2006)
30. Mutz, D., Valeur, F., Vigna, G., Kruegel, C.: Anomalous System Call Detection. ACM Transactions on Information and System Security **9**(1) (February 2006) 61–93
31. Crandall, J.R., Su, Z., Wu, S.F., Chong, F.T.: On Deriving Unknown Vulnerabilities from Zero-Day Polymorphic and Metamorphic Worm Exploits. In: Proceedings of the 12^{th} ACM Conference on Computer and Communications Security (CCS). (November 2005)
32. Anonymous: Anonymized. In: Technical Report