Transparent Web Service Auditing via Network Provenance Functions *

Adam Bates, Wajih Ul Hassan
University of Illinois at Urbana-Champaign
{batesa,whassan3}@illinois.edu

Kevin Butler, Alin Dobra, Bradley Reaves
University of Florida
{butler,adobra,reaves}@ufl.edu

Patrick Cable, Thomas Moyer, Nabil Schear
MIT Lincoln Laboratory
{cable,tmoyer,nabil}@ll.mit.edu

ABSTRACT

Detecting and explaining the nature of attacks in distributed web services is often difficult – determining the nature of suspicious activity requires following the trail of an attacker through a chain of heterogeneous software components including load balancers, proxies, worker nodes, and storage services. Unfortunately, existing forensic solutions cannot provide the necessary context to link events across complex workflows, particularly in instances where application layer semantics (e.g., SQL queries, RPCs) are needed to understand the attack. In this work, we present a transparent provenance-based approach for auditing web services through the introduction of *Network Provenance Functions* (NPFs). NPFs are a distributed architecture for capturing detailed data provenance for web service components, leveraging the key insight that mediation of an application's protocols can be used to infer its activities without requiring invasive instrumentation or developer cooperation. We design and implement `NPF` with consideration for the complexity of modern cloud-based web services, and evaluate our architecture against a variety of applications including DVDStore, RUBiS, and WikiBench to show that our system imposes as little as 9.3% average end-to-end overhead on connections for realistic workloads. Finally, we consider several scenarios in which our system can be used to concisely explain attacks. `NPF` thus enables the hassle-free deployment of semantically rich provenance-based auditing for complex applications workflows in the Cloud.

Keywords

Security; Audit; Data Provenance

1. INTRODUCTION

An increasing fraction of web and computing services run on cloud service platforms such as Amazon AWS, Google GCE, or Microsoft Azure [13]. These cloud service providers offer an ecosystem that allows customers to rapidly build sophisticated applications by interconnecting myriad software artifacts including storage, load balancers, and proxies. Many of these components are published by third-party developers, or may even be administrated as a proprietary service by another company. While dramatically simplifying deployment, this approach makes it difficult to reason holistically about the nature of suspicious events within an application. Each component maintains its own audit log, or performs no auditing at all. Moreover, due to the proprietary or third party nature of these software artifacts, modifying a component to improve its auditing capability may be out of the question.

Data provenance describes the history of the execution of a computing system, providing detailed explanations as to how data objects were created and came to arrive at their present state. Recently, provenance has been demonstrated to be of immense value in performing forensic reconstruction of a chain of events following an attack [3, 16, 28]. For example, provenance can indicate which hosts, processes, files, and data have been affected during an attack, and cue cleanup and recovery [24]. In addition, provenance is also a valuable tool in benign circumstances where a context-sensitive decision must be made about a data object or workflow.

While data provenance seems like a natural choice for addressing the challenges of web service auditing, there is not at this time a domain-general solution for deploying provenance tracking in existing applications. While a variety of tools to aid in the design of provenance-aware applications exist [12, 16, 18, 28], they are not well-suited to large applications because they assume full developer cooperation or access to source code. Provenance-aware operating systems [5, 9, 10, 19, 21, 22] provide automated and non-intrusive solution to capture provenance, but they suffer from semantic gap problems [23] in that the information they collect is too coarse-grained to satisfactorily explain application layer behaviors. For example, while an unauthorized data access in a web service will involve reading certain sensitive records from a database, the operating system's view of this same event will be a file access that is indistinguishable from legitimate data accesses. For provenance-based auditing to be viable in modern web services, what is required is a non-invasive approach to extracting semantic context to application layer software components.

In this work, we introduce the notion of Network Provenance Functions (`NPF`s), a proxy-based approach to deploying provenance-based auditing that is *general*, *low cost*, and *immediately deployable*. `NPF` transparently interposes on communication between different web service components to infer their activities. Our approach leverages a key insight that, while web services are growing increasingly sophisticated, their components are interconnected based on a manageable number of publicly available communica-

*The Lincoln Laboratory portion of this work was sponsored by the Assistant Secretary of Defense for Research & Engineering under Air Force Contract #FA8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are those of the author and are not necessarily endorsed by the United States Government.

WWW 2017, April 3–7, 2017, Perth, Australia.
ACM 978-1-4503-4913-0/17/04.
http://dx.doi.org/10.1145/3038912.3052640

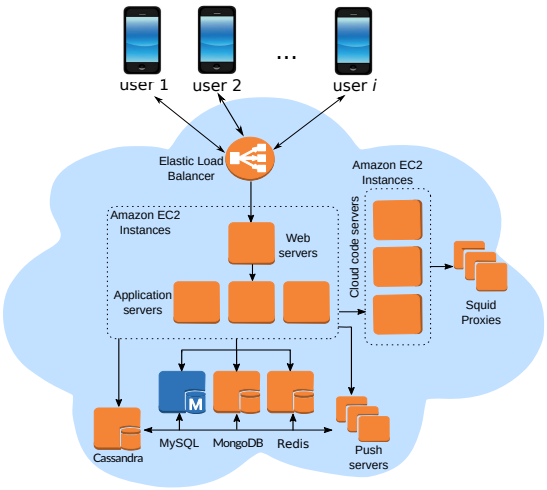

Figure 1: Diagram of a Parse web service architecture (simplified). To accurately track provenance, it is necessary to track individual client requests from the network, through the server, to the database, and back

tions protocols (e.g., HTTP, SQL, SOAP) and data marshaling standards (e.g., JSON, XML). For example, if a web server is communicating with a database, an NPF can intercept and interpret the communication to infer the relationship between remote web requests and database accesses. NPF thus provides a holistic view of system activity in large, complex computing workflows.

Our contributions can be summarized as follows:

- **Network Provenance Functions**: We present the design of NPF, and explore the space of auditing challenges that can be solved via our approach. NPF is comprised of an explicit forward proxy server, protocol parsers, and a data model for representing different forms inter-component communication.
- **Implementation & Evaluation**: We implement NPF and benchmark our system to demonstrate that NPF imposes as little as 9.3% overhead on web requests, which can be easily provisioned for in a cloud-based environment, and can be queried to determine the provenance of individual messages in just 1.23 ms.
- **Usage Demonstrations**: Through a series of case studies, we demonstrate how NPF can be employed to reasoning about Internet-based attacks. For instance, we demonstrate that our system can be used as a means of explaining SQLi-based data exfiltration, one of the most widespread threats to the Internet today.

2. MOTIVATION

Cloud Computing has revolutionized the way web services are designed and deployed today. With the *pay-as-you-go* and *provision-as-you-go* model, developers can deploy their web services in clouds such as Amazon Web Services (AWS) with greater scalability and flexibility. However, this increasing intricacy also complicates the task of diagnosing anomalous behavior, be it malicious activity or benign misconfigurations. Unfortunately, the primary focus of existing security solutions is to provide monitoring at the network boundaries (e.g., firewalls, network IDS), leaving them unfit to monitor violations that occur within the web service interior.

To ground our discussion of modern web services, we now describe the Parse service as an exemplar. Parse is a Facebook-owned business that provides cloud-based backend services to over 180,000 mobile applications. They provide a description of their architecture as part a testimonial on the AWS website.[1] A simplified example of Parse's virtual architecture is shown in Figure 1. Parse simultaneously serves two stakeholders: their clients (app developers), and their clients' users. When a user issues a request to a Parse-based app, the request passes through a load balancer, web server, and application server, which prompts queries to storage backends such as MySQL and MongoDB, and may also trigger additional messages be sent to a variety of push servers. Simultaneously, Parse clients use a cloud-based environment for hosting and developing their code, which can also interact with the main application servers (e.g., app updates) and is performance-optimized through use of a code proxy.

Consider a scenario in which a SQL injection (SQLi) based data exfiltration attack is launched on a Parse-based application: 1) the attacker sends a web request containing a malcrafted application input, 2) the request is directed to a web server by the load balancer, 3) the web server passes the malicious payload to an application server, 4) the application fails to properly sanitize the adversary-controlled input, leading to an unauthorized query "`SELECT * FROM CUSTOMERS`" which is then 5) returned to the attacker in the application's response. Later, after being the subject of an embarrassing news article about the data leak, an administrator wishes to understand how this attack took place and who was responsible.

A complete causal reconstruction of the attack is comprised of a) the remote source that initiated the attack, b) the worker instances that interacted with attacker input, and c) the data accessed in storage. Unfortunately, these vital pieces of information are spread out across the workflow: the source of the attack is obscured after the passing through the web server component, the instances that handled attacker input are determined dynamically by the load balancer and by a pool of database connections, and the unauthorized data access is not salient until *after* the malcrafted input passes through the application worker. As a result, a holistic explanation of such an attack eludes modern web service deployments. Network security monitoring at the boundaries of the architecture is oblivious to the attack as the violation occurs at the application layer, and those components that generate their own audit logs lack the means to fuse this information into a global view of the system's execution.

2.1 Provenance-based solutions

Our work is part of a growing body of literature that explores the use of data provenance to address critical security challenges. Provenance has been employed to detect compromised nodes in data centers [3, 8, 24, 28], explain and prevent data exfiltration [5, 14], and enrich access controls [4, 20]. Due to space limitations, we are unable to provide a full background on data provenance; a number of example provenance graphs are included in Section 6. Below, we consider the shortcomings of past approaches to provenance collection within the context of this domain.

While past work provides partial solutions to the web application problem, significant challenges remain to deploying provenance in this environment [2]. By deploying a provenance-aware operating system such as ProTracer [17] or LPM [5], it would be possible to use a single capture agent to observe all activities. Unfortunately, system layer provenance will be of little value alone due to the semantic gap between the OS and the application layer. Database primitives such as tables, records, and columns would not be identifiable in the provenance record. Moreover, the inability to distinguish between web requests at the server would create a *depen-*

[1] See `https://aws.amazon.com/solutions/case-studies/parse/`

dency explosion problem, leading to the false conclusion that each server response was dependent on all previous client requests.

An alternative approach would be to deploy provenance-aware applications. Although provenance libraries exist to simplify the manual instrumentation of source code [18], this approach requires additional capital and domain-specific knowledge that is unlikely to be available to most web developers. Furthermore, the developers may be forced to instrument not only their own code, but also many of its dependencies, including the web server, runtime framework, database service, and 3rd party libraries that they use. Tools such as BEEP can be used to partially automate the creation of provenance-aware applications [16]. Unfortunately, while BEEP addresses the dependency explosion problem, a semantic gap still exists because BEEP provenance is extracted through the use of system layer audit logs. To observe the database, we could turn to dedicated provenance-aware database management services such as Trio [25], DBNotes [7], and ORCHESTRA [15]. However, this solution requires re-architecting the web service, and does not provide linkability between the service and the database. Furthermore, while the means of deploying provenance capabilities in databases have been known for over a decade, there has been little to no adoption by mainstream database services, indicating that database developers are uninterested in supporting this functionality.

These proposed approaches suffer from two fundamental limitations: (1) they cannot observe information flows beyond the boundaries of their own operation; and (2) they require significant changes to the OS and applications. With this in mind, the broad challenge we consider is the development of a minimally invasive and cost-effective means of creating provenance-aware web services. Our solution, `NPF`, provides provenance-based auditing without requiring modification to existing web applications and databases.

3. DESIGN

3.1 Threat Model & Assumptions

The attack surface we consider in this work is that of a typical cloud-based web service. By connecting to the web service's external listening ports the attacker may attempt a variety of misdeeds on the system. The attacker may attempt to exfiltrate data from the web application through iterative command injection (e.g., SQLi) attacks. A successful exfiltration attack will involve repeated command injections as the attacker attempts to discover the location of valuable data. The attacker may also attempt to use command injection to inject spurious data into the database for the purposes of privilege escalation or cross-site scripting. Alternately, the adversary's target may not be the web application but the system itself. The attacker may be targeting the web server in order to compromise other services on the host or to move laterally to other hosts on the network [27].

We make the following assumptions about the security of each web service component inside the cloud. We conservatively assume that the components of the web service are all subject to compromise. These components may begin to lie about their actions on the system at any time, but we assume that at least one record of the attacker's access attempt is recorded prior to compromise.

3.2 Design Goals

G1 Complete. Our system must offer a complete description of individual requests as they pass through the web service workflow. The record must remain complete in the presence of unexpected events triggered by attacker behavior, such as command injection attacks or binary exploitation.

G2 Integrated. Our system must combine provenance from different software components in a salient manner that provides a coherent explanation of application activity to the administrator. Provenance generated by different audit mechanisms must share a common namespace, and each audit mechanism must be able to accurately reference the activities of other agents.

G3 Widely Applicable. To further advocate for the deployability of provenance-aware applications, our efforts in the development of the system should not be limited to the benefit of a particular application, backend component, or architecture. Instead, our system should be immediately compatible with a broad number of existing applications.

3.3 Network Provenance Functions

A fundamental design consideration in our system was the manner in which our auditing mechanism would observe communication between system components. One possibility would be to instrument the components to add auditing capabilities. However, this would have made our solution application-specific, and more generally would be at odds with the rapid deployment model of modern web services. Instead, we chose to create the `NPF` auditing mechanism in the form of explicit forward proxies that monitor on communications between system components. A key insight to this approach is that, while web services are growing increasingly sophisticated, their components are interconnected based on a manageable number of publicly available communications protocols (e.g., HTTP, SQL, SOAP) and data marshaling standards (e.g., JSON, XML). By parsing and interpreting these communications, it is possible to infer the internal state of the software component, without requiring any modifications to the component itself. Below, we describe the two primary components of `NPF`– protocol parsing §3.3.1 and audit event generation §3.3.2.

3.3.1 Protocol Parsers

After proxying the inter-component traffic, we make use of protocol grammars in order to infer the components' actions and subsequently generate provenance records. Each grammar accepts as input a protocol message and outputs a parse tree representing the syntactic structure of the message. Following construction of the parse tree, each `NPF` protocol parser defines an algorithm that traverses the parse tree and extracts the semantically-relevant pieces of a protocol message. Below, we consider two common web service protocols – Structured Query Language (SQL) and Simple Object Access Protocol (SOAP).

Structured Query Language (SQL): As an exemplar database query language, we focus on SQL due to its widespread use. After specifying a SQL grammar, we obtain a SQL query parse tree such as the one shown in Figure 2. This tree is a SELECT statement that contains a FROM clause (required) and a WHERE clause, which is one of several optional SQL clauses. The goal of our SQL parser is then to extract two important classes of information from each query – the access type and the data objects. The access type is determined by the root node of the parse tree (e.g., `select_stmt`. The data objects are contained in the leaves of the parse tree; however, the canonical name for most data objects contain both a table and a column reference, which are found in different clauses of the statement. To address this, we defined a synthesized attribute to aggregate this information at the root of the tree. As the tree is constructed, the attribute tracks the columns used in the current subtree and the tables used in the FROM clause. When the parse tree is complete, the synthesized attribute is inspected to determine the appropriate prefix for each column given the tables used in the FROM clause.

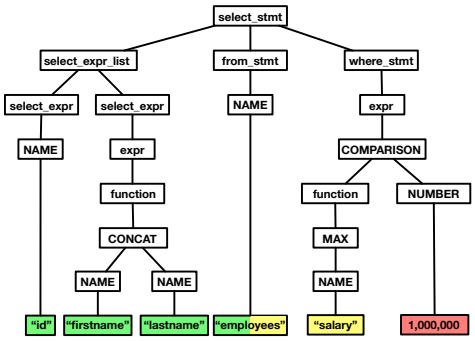

Figure 2: A SQL parse tree for the statement "`SELECT employee_id, CONCAT(firstname, lastname) FROM employees WHERE MAX(salary) > 1,000,000`".

In non-trivial SQL statements, a variety of challenges arise in extracting this data. We came across a number of such challenges while designing and implementing `NPF`. We describe our solutions to each problem below:

- *Parsing Challenge #1: Wildcards*. Through use of the wildcard character, SQL statements are able to reference all columns in a table without explicitly naming them. To address this, we provide the `NPF` with a schema description that allows it to translate the wildcard character into the associated columns for the given table. In our implementation, we obtain the schema through use of the `mysqldump` command.

- *Parsing Challenge #2: Aliases*. Any value in a SQL statement can be aliased to another name. The challenge in resolving aliases is that an alias may be referenced in one clause of the query, but defined in another. To address this problem, we extended our synthesized attribute to track references and definitions of aliases. At the top level of the parse tree, the list of referenced aliases are then resolved to their true table and column names. In effect, this means that `NPF` unaliases named objects during parsing, ensuring that the extracted provenance is unobfuscated.

- *Parsing Challenge #3: Nested Queries*. An additional obstacle we faced in the design of our grammar was that of nested queries. In SQL, full statements can be indefinitely nested within one another. For example, "`SELECT A FROM (SELECT id AS A FROM employees)`" is a valid statement. Nested queries can be used to further obfuscate the true origin of a data object. Our solution to this is to modify the synthesized attribute routines described above. At the root of each subquery in the parse tree, objects in the subquery are unaliased, and the named and referenced objects used by the subquery are transferred to the parent query. Additionally, the alias mapping is passed to the parent query, allowing `NPF` to unnest queries as the statement is parsed.

Simple Object Access Protocol (SOAP): Remote procedure call frameworks allow one application to invoke functions in other applications. As a canonical example, we consider the SOAP specification. SOAP encodes the requests and responses in XML, and can be used over a variety of transports, including HTTP, SMTP, and standard sockets. A SOAP request includes the method name to be invoked on the remote system and the parameters for the method. The response includes the result of the method invocation or an error message. For each request response pair, we capture the method

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>
    <m:GetEndorsingBoarder xmlns:m="http://namespaces.snowboard-info.com">
      <manufacturer>K2</manufacturer>
      <model>Fatbob</model>
    </m:GetEndorsingBoarder>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 3: Example SOAP request from `https://www.w3.org/2001/03/14-annotated-WSDL-examples.html`

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>
    <m:GetEndorsingBoarderResponse xmlns:m="http://namespaces.snowboard-info.
        com">
      <endorsingBoarder>Chris Englesmann</endorsingBoarder>
    </m:GetEndorsingBoarderResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 4: Example SOAP response from `https://www.w3.org/2001/03/14-annotated-WSDL-examples.html`

name, the input arguments, and the response. Figure 3 shows an example request invoking the `GetEndorsingBoarder` method with two arguments, `manufacturer` and `model`. The response in Figure 4 is another SOAP envelope containing `endorsingBoarder` key with value "Chris Englesmann".

- *Parsing Challenge #4:* The SOAP specification does not define the content of the message body beyond requiring that the body be valid XML. This makes it difficult to determine what the method name and arguments are for a given message. For application developers, a description of the services are provided via a WSDL[2] file, an XML file that defines valid methods and the required arguments for each method. We provide the SOAP `NPF` with the WSDL for the host, allowing the parser to extract valid method names and what parameters each method expects.

3.3.2 Audit Event Generation

Below, we describe how `NPF` generates audit events across for database accesses (e.g., SQL) and remote procedure invocations (e.g., SOAP). The event streams from different `NPF`s are received at the Provenance Recorder and then processed into provenance graphs of system execution. As the process of graph generation is based on open specifications [26], for brevity we have omitted a description of this process. Example provenance graphs can be found in Section 6.

Database Accesses

- *Explicitly Accessed Data:* We refer to the named objects referenced in the primary clause of the query as accessed data. These are the objects that will be returned by the database in its response to the query. `NPF` generates a `Used` provenance relation for each piece of accessed data in SELECT, SHOW, and DESCRIBE clauses; for INSERT and UPDATE clauses, `WasGeneratedBy` relations are generated for each data access. The accessed data in Figure 2, shaded in green, are the *employee_id*, *firstname*, and *lastname* columns, and the *employees* table.

- *Implicitly Accessed Data:* Named objects that appear in subsequent clauses of the query are not explicitly returned by the database, but nonetheless inform the response message. Consider the implicit data access shaded in yellow in Figure 2 – while

[2]Web Services Description Language

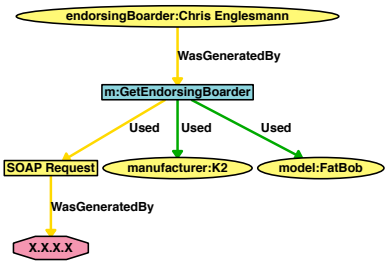

Figure 5: An example of a provenance graph for a SOAP response, showing the method, input arguments, and the requesting client `X.X.X.X`.

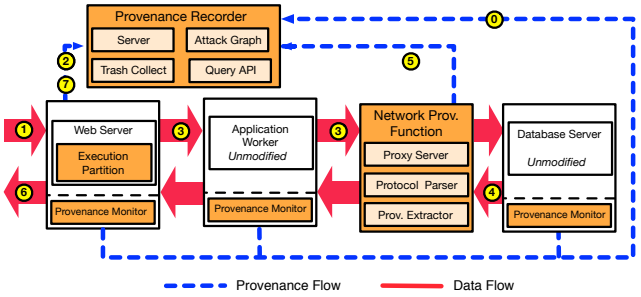

Figure 6: Diagram of a web service deployment with `NPF`. The introduced components are shaded in orange. No changes are required to the Database Engine or Web Application; instead, provenance is generated by interposing on inter-component communications, with supplemental audit records provided by system layer provenance monitors. A small change to the Web Server is required to partition incoming requests.

employee salaries are not returned in the query, the response implicitly informs the querier of which employees salaries are greater than $1,000,000. Referenced data therefore represents a dangerous side channel for information leakage. However, in some environments, it may be unnecessarily conservative to treat all referenced data as accessed data. To account for this, we introduce a `USED_IMPLICIT` event to describe referenced data.

- *Access of Ephemeral Data:* Query expressions also include non-persistent data objects, such as numbers and string literals. Ephemeral data can manipulate the records and values returned by a query, but not the columns accessed and referenced. We choose to ignore ephemeral data for the case of audit event generation, such as the $1,000,000 constant shaded in red in Figure 2.

Remote Procedure Invocations

- *Methods:* Unlike the SQL `NPF`, where the database management server activity is implicit, the SOAP method defines an `Activity`. The method name is extracted from the SOAP message using the WSDL to determine the validity of the method name. Valid activities are reported to the provenance recorder, along with the inputs and outputs of the method.

- *Inputs/Outputs:* One challenge associated with parsing SOAP messages is determining the arguments sent to the RPC end-point, and the response returned to the client that invoked the method. The WSDL provided by the end-point is used by the SOAP `NPF` to determine the names of the method arguments and responses. The method arguments are extracted from the SOAP message, and each input argument is associated with the RPC activity with a `Used` provenance relation. In Figure 3, the `manufacturer` and `model` data objects are linked to the `GetEndorsingBoarder` activity. For responses, a `wasGeneratedBy` provenance relation is created for each element of the response. In our example, the `endorsingBoarder` response element is linked to the method `GetEndorsing-Boarder`. Figure 5 shows the provenance graph that is generated from the request-response pair from Figures 3 and 4.

3.4 Supplemental Tracking Techniques

In addition to `NPF`, we take note of two important classes of audit information that require additional context in order to track. First, in the event that a software component such as the web server is compromised, `NPF` alone is insufficient to track attacker actions. This is because the attacker will no longer be limited to the workflow activities, but will instead be able to take any action on the system with the privileges of the web server. To ensure the ability to track attacker actions in the presence of system layer attacks, we run our software components on top of provenance-aware operating systems. We make use of a system layer provenance monitor such as Linux Provenance Modules [5] or SPADE [10] Second, at points in the workflow where multiple client connections are multiplexed on a single worker instance (e.g., web server), `NPF` encounters a *dependency explosion* problem in which the inputs and outputs from the component cannot be disambiguated, leading to overly conservative and false dependency assumptions. To address this, we make use of execution partitioning techniques as introduced in [16]. Both LPM and Execution Partitioning leads to the emission of additional provenance events that are aggregated at the recorder and synthesized into the attack graph.

We note that the use of these techniques does not impact the adoptability or generality of our system. LPM is implemented for the Red Had Linux kernel and can be packaged as an EC2 image in order to avoid deployment costs. Execution Partitioning will involve a minimal modification to the web server that we discuss in Section 4. However, the vast majority of web services are based on just a few servers such as Apache and Nginx, making instrumentation a one-time application-agnostic effort. Our architecture thus maintains its generality in spite of these extensions.

3.5 Deployment

We now demonstrate a possible deployment scenario for `NPF` and step through how an individual web request is processed. A diagram of a simple `NPF` deployment is shown Figure 6 featuring a Web Server, Application Worker instance, and Database Server. The original data flow of the web service is marked by red arrows, while the stream of provenance events is marked by blue dashed lines. `NPF` introduces several components (shaded in orange), but requires no modification to the application or database management system. The added components are as follows: a small and re-usable modification to the web server that performs execution partitioning on incoming requests, an `NPF` mediating communication between the application worker and the database, and a system layer provenance monitor. All of these components generate provenance events that are transmitted to a central provenance recorder described in Section 4.

The workflow for provenance collection is as follows: (1) a remote host makes a request to the Web Application; (2) a small modification to the web server notifies the Provenance Recorder that a new autonomous unit of work has begun; (3) `NPF` proxies and parses a SQL query issued to the Database Server; (4) `NPF` observes the Database Server response and (5) reports new prove-

nance events to the Provenance Recorder (6) after the Web Applications transmits a response to the remote host, (7) the Web Server notifies the Provenance Recorder that the unit of work has ended; (0) throughout this process, the system-layer provenance monitor generates provenance for all activities that are not being explicitly observed by NPF.

3.6 Security Analysis

Our arguments for satisfying the Design Goals are as follows:

Complete (G1). NPF is able to differentiate between individual web requests through a minimal modification to the Web Server that performs execution partitioning. Transactions between individual components are observed by the NPF proxy. In the event that the server is compromised, NPF can continue to track the actions of the attacker on the system through use of the provenance-aware kernel, whose trusted computing base can be protected using SELinux enforcement [5].

Integrated (G2). In NPF, we integrate provenance from different protocols through tracking host and process ids (pid). In our implementation, the server components are configured to run in pre-fork mode in which all workers receive a unique pid. When a request is proxied by NPF, the sending component is identified at the Provenance Recorder by matching the network context (i.e., IP Addr, Port) between the component and the NPF proxy. The host and process id's are then linked to all provenance events reported on their actions. To integrate NPF provenance with system-layer provenance, we introduce a `pid_to_provenance` syscall that accepts a pid and returns the universally unique identifier associated with the provenance of the process' fork in the kernel. Thus, all layers share a common language to describe an activity.

Widely Applicable (G3). We confirmed that our NPF implementation is compatible with the Apache 2 and Tomcat (via mod_jk) Web Servers and a variety of web applications and utilities including MediaWiki[3] (PHP-based), UnixODBC[4], and a suite of Tomcat-based applications that were released with the AMNESIA [11] evaluation testbed[5]. While in practice SQL comes in many different flavors and varies by database management system, we found that our method of identifying data objects was simple enough to work on a variety of SQL platforms including MySQL and PostGreSQL.

4. IMPLEMENTATION

Network Provenance Functions: We have implemented NPF in C as a multithreaded TCP proxy server. The SQL Protocol Parser uses Bison and extends a publicly available SQL grammar[6] with a synthesized attribute that identifies all accessed data objects as described in §3.3.1. After parsing the query, NPF inspects the list of data objects accessed. It then creates a provenance event for each object, which is a tuple of the form $< pid, relationship, column, table >$. $Relationship$ is one of the several relationships specified in Section 3.3, and are also consistent with the Open Provenance Model (OPM)[7] While we have focused on using NPF with SQL and SOAP, NPF for SQL protocol but our system can be extended to any web service communication such as HTTP or REST APIs by defining parsers for those protocols.

[3] Available at `https://www.mediawiki.org`.

[4] Available at `http://www.unixodbc.org`.

[5] Available at `http://www-bcf.usc.edu/~halfond/testbed.html`.

[6] Available at `https://github.com/hoterran/sqlparser`.

[7] See `http://openprovenance.org/`

Benchmark	Total Queries	Database Size (GB)	Average Time (ms) w/o NPF	Average Time (ms) with NPF	Percent Overhead
Dell DVD Store	6451	10	10.7	11.7	9.3
RUBiS	6430	1	6.5	7.2	11.2
WikiBench	6581	3	6.3	7.0	11.6

Table 1: Performance of NPF under realistic workloads.

Provenance Recorder: The Provenance Recorder is responsible for aggregating provenance from the different NPFs and representing it in an query-efficient in-memory graph. We implemented the Recorder in C++ using the SNAP graph library. Different provenance events are handled as described in Section 3.3; generally speaking, when the Recorder receives a new event it first checks to see if any of the involved objects are already present in the graph, creates them if they are not, and then adds a new relationship between the objects. Visual examples of how provenance graphs were represented by the recorder follow in Section 6.

Provenance-Enhanced Web Server: We instrumented Apache2 (version 2.2.31) to perform execution partitioning within the the `ap_invoke_handler` function. We selected this layer of the server stack because it after the various error and security filters performed by the server, ensuring that we do not generate provenance for requests that are later rejected, but before file-specific handler modules, providing a single point of mediation for monitoring both static and dynamic content. We first configured the server to run in pre-forked mode.[8] We then included an additional header file in the Apache 2 source to handle the generation and transmission of provenance events. Finally, we inserting two function calls in `server/config.c`, before and after the call to `ap_run_handler`. Before `ap_run_handler`, the `npf_unit_start` function generates a UUID to associate with the unit of work, then transmits the message to the Provenance Recorder that contains the UUID, the `remote_addr` struct, and the process id of the Apache worker. After `ap_run_handler`, the `npf_unit_end` function transmits a message to the Provenance Recorder that notifies the recorder that the unit of work associated with UUID has ended, then frees the UUID character array. This approach to execution partitioning is general enough to work with any other C-based server implementation such as *Nginx* and *sshd*.

System-Layer Provenance Monitor: To ensure the ability to track attacker actions after a binary server exploit, we provision all software components on top of a provenance-aware operating system. We made use of the Linux Provenance Modules project for our provenance-aware kernel. We configured LPM to make use of the Hi-Fi module [21], and deployed the system as described in [5].

5. PERFORMANCE EVALUATION

We conducted our evaluation on a machine with two 2.4 GHz Quad-Core Intel Xeon processors and 12 GB RAM. VMWare Fusion VMs running CentOS 6.5. VMs were provisioned with VMWare Fusion using common deployment model of 4GB RAM and 2 vCPUs and ran CentOS 6.5. Due to space limitations, we include tests only for the SQL NPF below.

End-to-End Delay One of the vital measures of NPF's performance is the end to end delay it imposes on web requests. We represented realistic workloads to evaluate NPF using three benchmark tests: the Dell DVD Store e-commerce site,[9] RUBiS auction

[8] See `http://httpd.apache.org/docs/2.2/mod/prefork.html`.

[9] Available at `http://linux.dell.com/dvdstore/`

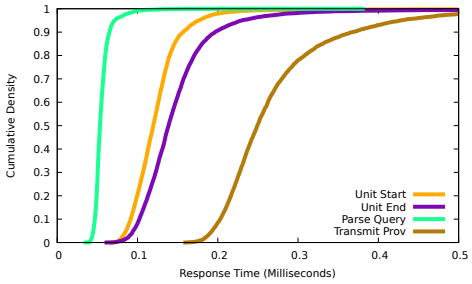

Figure 7: CDFs for NPF microbenchmarks during DVDStore test.

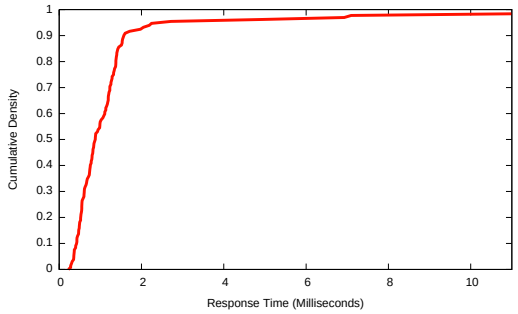

Figure 8: CDF of Provenance Recorder's query response time.

site,[10] and the Mediawiki-based WikiBench.[11] DVDStore was run using a MySQL that was populated with 10 GB of content using the DVDStore data generation programs. RUBiS was run using a database acquired from [6] containing 9 tables containing 33,721 items for sale in 20 categories, and traffic was generated using the RUBiS client emulator. Mediawiki was populated using a trace file from September 27th[12] and traffic was generated by issuing cURL requests for random wiki pages.

Each of the above tests was run twice, both with and without NPF interposition. The results are shown in Table 1 When averaged over thousands of connections in each test, NPF imposed between 0.7 and 1 ms overhead per connection, which represents 9.3% – 11.6% overhead due to the high throughput of the applications. Overhead did not increase with the size of the database. While 10% overhead is non-negligible, we conclude that NPF's performance falls within acceptable bounds for real applications.

Microbenchmarks To determine the cause of this overhead, we instrumented NPF to measure the average time spent on each individual step during experimentation. Specifically, we measured the average time spent in the SQL parsing step, transmitting provenance from NPF to the Provenance Recorder, transmitting unit_start and unit_stop to the Provenance Recorder. Figure 7 shows the associated cumulative density functions for each measured step during DVDStore benchmark. The various SQL queries generated by DVDStore could be parsed and have their provenance extracted in an average of 0.053 ms. The primary source of delay in our NPF system is due to inter-process communication; transmitting provenance events required 0.318 ms on average, and 7.624 ms in the worst case. unit_start and unit_stop messages, which are smaller than messages from the NPF, transmitted in an average of 0.137 ms and 0.172 ms respectively. Steps that required communication with the Provenance Recorder experienced high variance, indicating processing delays at the Recorder that could be addressed to improve performance. As our provenance recorder implementation was single-threaded, it is likely that delays could be dramatically reduced through creating a multi-threaded version of the recorder.

Provenance Query Performance A final performance consideration for NPF is the speed with which provenance data can be queried by an administrator. The ability to quickly access relevant audit data is of critical importance when responding to live security incidents. In this test, we then issued a series of queries to the Provenance Recorder using the Query API. We repeated the DVDStore benchmark, and requested the ancestry of each server response to the DVDStore Client immediately after the server transmitted a unit_end message, producing provenance graphs similar to Figure 9 The results are shown in Figure 8. The average response time by the Provenance Recorder was just 1.23 ms. We microbenchmarked this result as well, and found that on average 1.17 ms of this delay was due to IPC, while just 0.5 ms was required to generate the provenance ancestry. In the worst case, the query took 7 ms to respond, but this was also due to IPC delays and not to the cost of graph traversal. These results indicate that our implementation of NPF is already able to quickly respond to forensic queries, and is so fast that it may even be used as the basis for automated provenance-based enforcement mechanisms.

[10] Available at http://rubis.ow2.org/
[11] Available at http://www.wikibench.eu/
[12] Available at https://archive.org/details/enwiki-20080103

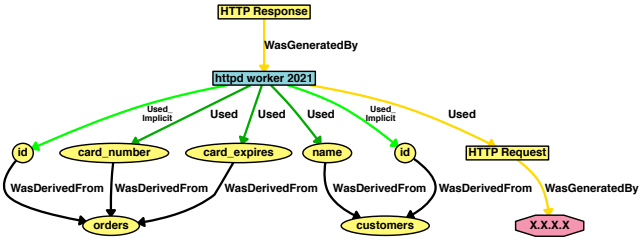

Figure 9: The provenance of a successful SQL injection attack on an eCommerce site launched by remote host X.X.X.X. The attack accesses sensitive data from the customers and orders tables.

6. CASE STUDIES

In this section, we consider several attack scenarios in which NPF can be used to detect and explain Internet-based attacks.

6.1 Scenario #1: SQL Injection

One of the primary motivations in the design of NPF was to be able to track common web application vulnerabilities such as SQL Injection (SQLi). SQLi enable data exfiltration attacks on large company customer databases and are thus one of the most prevalent threats on the Internet today. However, since a full explanation of SQLi involves a composition of network context, bypassed application logic, and database accesses, the ability to succinctly explain SQLi attack paths eludes modern audit system. In this scenario, we simulated a successful SQLi attack by using a toy PHP application on Apache 2 that relayed SQL queries to MySQL over POST requests; this behavior is similar to an attack in which sanitization checks within the web application are successfully bypassed, leading to a valid (but malicious) SQL query

A condensed version of the attack graph recorded by NPF is shown in Figure 9. Here, we see the provenance of a message derived from SQL data from the customers and orders tables. Critically, the query obfuscations that are commonly associated with SQLi are not present in the provenance graph. This is because such

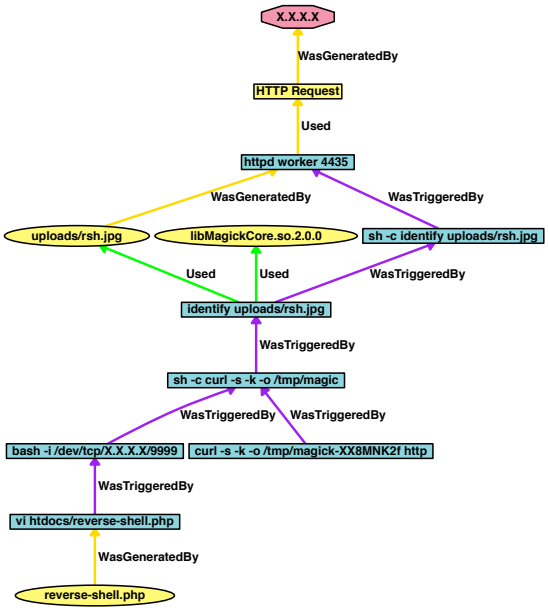

Figure 10: The provenance of an ImageTragick exploit. The attacker uploads a malicious image file `rsh.jpg` that opens a remote shell back to the attackers host. The attacker gains persistence on the machine by placing a reverse shell in the server's `htdocs` folder.

obfuscations are designed to bypass input sanitization checks that are performed by the web application. When a malicious input is able to successfully pass through these checks, the output is a well-formed SQL query. As a result, NPF is optimally positioned to understand the intent of the attacker. Moreover, while our motivating usage scenario has been on forensic (i.e., offline) investigation, the information collected by NPF could also be used to perform real time notification and incident response. For example, an administrator could specify that credit card numbers are only permitted to flow to remote IP addresses associated with the Visa Payment Network, then perform realtime filtering of HTTP responses using the provenance captured by NPF.

6.2 Scenario #2: ImageTragick Exploit

To demonstrate the combined capabilities of NPF and system layer monitoring when deployed in tandem, we developed a web application exploit based on the recently discovered vulnerabilities in the ImageMagick library.[13] Our vulnerable web application made use of a PHP script that used the ImageMagick to test whether an uploaded file was an image. The attack payload was a `jpg` comprised of 4 lines of text including the following command:

```
image over 0,0 0,0 'https://127.0.0.1/x.php?x=`bash -i
    >\& /dev/tcp/X.X.X.X/9999 0>\&1`'
```

This code is executed by the server when the image is processed by the 'identify' ImageMagick tool, causing a bash shell to be linked to the attackers remote host on port 9999. The provenance of these activities on the server side is shown in Figure 10. Without NPF, the operating system provenance for these events would be difficult to interpret. Dependency explosion would make it hard to identify which remote host was able to invoke a shell command. NPF signals the start of a unit of work before the request is handled,

[13]See https://imagetragick.com/

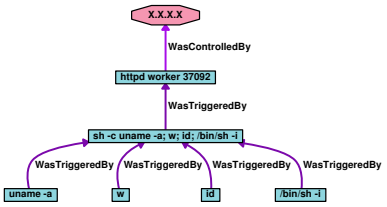

Figure 11: The union of NPF and system layer provenance can track a remote shell invocation launched by remote host `X.X.X.X`. The provenance of file and packet manipulations have been pruned for clarity. NPF assists the forensic process by identifying the remote host and unit of work responsible for the exploit.

which removes from suspicion all sessions that occurred prior to the compromise. Following the compromise, LPM can be securely configured such that the attackers actions on the system can continue to be monitored [5].

6.3 Scenario #3: Reverse Shell Invocation

As a final example, we monitor the attacker's subsequent visits to the web server through invoking the `reverse-shell.php` script. To create realistic attack provenance, we made use of a publicly available php-reverse-shell application.[14] Reverse shells were also an aspect of the Apache 2 Darkleech attack [1]. Figure 11 shows the resulting provenance graph. Here, we can see that an `httpd` worker with pid 37092 is handling a request from remote host `X.X.X.X`. Unexpectedly, the worker issues a series of conspicuous system commands that collect information regarding the name of the machine (`uname`), the identity of the active user (`id`) which in this case is a daemon, and the activities of other users that are currently logged in (`w`). The worker then drops to shell. This information would serve as an invaluable explanation as to the attackers intent once their intrusion had been discovered.

7. CONCLUSION

In spite of a pressing need for ways to explain and mitigate web application vulnerabilities, web services have received little attention as candidates for provenance-based auditing capabilities. In this work, we presented NPF, a system for creating provenance-aware web applications. Our system can be deployed without requiring any changes to the web application, yet provides rich, concise audit trails for web service workflows. We demonstrated NPF's ability to explain SQL injection attacks, and to aid in the tracking of system layer attacks against the web server. In evaluation, we discovered that our system imposes as little as 9% overhead on web requests. Thus, NPF demonstrates a deployment strategy for provenance-based auditing not only in web services, but for many complex, heterogeneous application workflows.

Acknowledgements

We would like to Mugdha Kumar for her assistance with the extension of Linux Provenance Modules. This work is supported in part by the US National Science Foundation under grant numbers CNS-1540216, and CNS-1540217.

Availability

Our code is available at `http://linuxprovenance.org`.

[14]Available at `http://pentestmonkey.net/tools/web-shells/php-reverse-shell`.

8. REFERENCES

[1] Darkleech Apache Attacks Intensify. http://www.darkreading.com/attacks-and-breaches/darkleech-apache-attacks-intensify/d/d-id/1109760?

[2] I. M. Abbadi, J. Lyle, et al. Challenges for provenance in cloud computing. In *TaPP*, 2011.

[3] A. Bates, K. Butler, A. Haeberlen, M. Sherr, and W. Zhou. Let SDN Be Your Eyes: Secure Forensics in Data Center Networks. In *SENT*, 2014.

[4] A. Bates, B. Mood, M. Valafar, and K. Butler. Towards Secure Provenance-based Access Control in Cloud Environments. In *CODASPY*, New York, NY, USA, 2013.

[5] A. Bates, D. Tian, K. R. Butler, and T. Moyer. Trustworthy Whole-System Provenance for the Linux Kernel. In *USENIX Security*, 2015.

[6] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Performance and scalability of ejb applications. In *ACM Sigplan Notices*, volume 37, pages 246–261. ACM, 2002.

[7] L. Chiticariu, W.-C. Tan, and G. Vijayvargiya. DBNotes: A Post-it System for Relational Databases Based on Provenance. In *SIGMOD*, 2005.

[8] A. Gehani, B. Baig, S. Mahmood, D. Tariq, and F. Zaffar. Fine-grained Tracking of Grid Infections. In *GRID'10*, 2010.

[9] A. Gehani and U. Lindqvist. Bonsai: Balanced Lineage Authentication. In *ACSAC*, 2007.

[10] A. Gehani and D. Tariq. SPADE: Support for Provenance Auditing in Distributed Environments. In *Middleware*, 2012.

[11] W. G. J. Halfond and A. Orso. Amnesia: Analysis and monitoring for neutralizing sql-injection attacks. In *ASE*, 2005.

[12] R. Hasan, R. Sion, and M. Winslett. The Case of the Fake Picasso: Preventing History Forgery with Secure Provenance. In *FAST*, San Francisco, CA, USA, 2009.

[13] K. He, A. Fisher, L. Wang, A. Gember, A. Akella, and T. Ristenpart. Next stop, the cloud: Understanding modern web service deployment in ec2 and azure. In *IMC*, 2013.

[14] S. N. Jones, C. R. Strong, D. D. E. Long, and E. L. Miller. Tracking Emigrant Data via Transient Provenance. In *TAPP'11*, 2011.

[15] G. Karvounarakis, T. J. Green, Z. G. Ives, and V. Tannen. Collaborative data sharing via update exchange and provenance. *ACM Trans. Database Syst.*, 38(3):19:1–19:42, Sept. 2013.

[16] K. H. Lee, X. Zhang, and D. Xu. High Accuracy Attack Provenance via Binary-based Execution Partition. In *NDSS*, 2013.

[17] S. Ma, X. Zhang, and D. Xu. ProTracer: Towards Practical Provenance Tracing by Alternating Between Logging and Tainting. In *NDSS*, 2016.

[18] P. Macko and M. Seltzer. A General-purpose Provenance Library. In *TaPP*, 2012.

[19] K.-K. Muniswamy-Reddy, U. Braun, D. A. Holland, P. Macko, D. Maclean, D. Margo, M. Seltzer, and R. Smogor. Layering in Provenance Systems. In *ATC*, 2009.

[20] J. Park, D. Nguyen, and R. Sandhu. A Provenance-Based Access Control Model. In *PST*, pages 137–144, 2012.

[21] D. Pohly, S. McLaughlin, P. McDaniel, and K. Butler. Hi-Fi: Collecting High-Fidelity Whole-System Provenance. In *ACSAC*, Orlando, FL, USA, 2012.

[22] C. Sar and P. Cao. Lineage File System. http://crypto.stanford.edu/~cao/lineage.html.

[23] M. Stamatogiannakis, H. Kazmi, H. Sharif, R. Vermeulen, A. Gehani, H. Bos, and P. Groth. Trade-offs in automatic provenance capture. In *IPAW*, pages 29–41, 2016.

[24] D. Tariq, B. Baig, A. Gehani, S. Mahmood, R. Tahir, A. Aqil, and F. Zaffar. Identifying the Provenance of Correlated Anomalies. In *SAC*, 2011.

[25] J. Widom. Trio: A System for Integrated Management of Data, Accuracy, and Lineage. Technical Report 2004-40, Stanford InfoLab, Aug. 2004.

[26] World Wide Web Consortium. PROV-Overview: An Overview of the PROV Family of Documents. http://www.w3.org/TR/prov-overview/, 2013.

[27] L. Zhang, A. Persaud, A. Johnson, and Y. Guan. Detection of stepping stone attack under delay and chaff perturbations. In *IPCCC*, pages 10 pp.–256, April 2006.

[28] W. Zhou, Q. Fei, A. Narayan, A. Haeberlen, B. T. Loo, and M. Sherr. Secure Network Provenance. In *SOSP*, 2011.