Asynchronous intrusion recovery for interconnected web services

Ramesh Chandra, Taesoo Kim, and Nickolai Zeldovich
MIT CSAIL

Abstract

Recovering from attacks in an interconnected system is difficult, because an adversary that gains access to one part of the system may propagate to many others, and tracking down and recovering from such an attack requires significant manual effort. Web services are an important example of an interconnected system, as they are increasingly using protocols such as OAuth and REST APIs to integrate with one another. This paper presents Aire, an intrusion recovery system for such web services. Aire addresses several challenges, such as propagating repair across services when some servers may be unavailable, and providing appropriate consistency guarantees when not all servers have been repaired yet. Experimental results show that Aire can recover from four realistic attacks, including one modeled after a recent Facebook OAuth vulnerability; that porting existing applications to Aire requires little effort; and that Aire imposes a 19–30% CPU overhead and 6–9 KB/request storage cost for Askbot, an existing web application.

1 Introduction

In an interconnected system, such as today's web services, attacks that compromise one component may be able to spread to other parts of the system, making it difficult to recover from an intrusion. For example, consider a small company that relies on a customer management web service (such as Salesforce) and an employee management web service (such as Workday) to conduct business, and uses a centralized access control web service to manage permissions across all of its services. The servers of these web services interact with each other on the company's behalf, to synchronize permissions, update customer records, and so on. If an attacker exploits a bug in the access control service, she could give herself write access to the employee management service, use these new-found privileges to make unauthorized changes to employee data, and corrupt other services. Manually recovering from such an intrusion requires significant effort to track down what services were affected by the attack and what changes were made by the attacker.

Many web services interact with one another using protocols such as OAuth [3], REST, or other APIs [12, 14, 22, 23], and several recent vulnerabilities in real web services [9–11, 13, 20] could be used to launch attacks like the one just described. For example, a recent Facebook OAuth vulnerability [9] allowed an attacker to obtain a fully privileged OAuth token for any user, as long as the user mistakenly followed a link supplied by the attacker; the attacker could have used this token to corrupt the user's Facebook data. If other applications accessed that user's data on Facebook, the attack would have spread even further. So far, we do not know of serious attacks that have exploited such vulnerabilities, perhaps because interconnected web services are relatively new. However, we believe that it is only a matter of time until attacks on interconnected web services emerge.

This paper takes the first steps toward automating recovery from such attacks. We identify the challenges that must be addressed to make recovery practical, and present the design and implementation of Aire, a system for recovering from intrusions in a large class of loosely coupled web services, such as Facebook, Google Docs, Dropbox, and Amazon S3.

Aire works as follows. Each web service that wishes to support recovery runs Aire on its servers. During normal operation, Aire logs information about the service's execution, as well as requests received from and sent to other services, thus tracking dependencies across services. When an administrator of a service learns of a compromise, he invokes Aire on the service, and asks Aire to cancel the attacker's request. Aire repairs the local state of the service using selective re-execution [7, 15], and propagates repair to other web services that may have been affected, so they can recover in turn, until all affected services are repaired. In addition to recovering from attacks, Aire can similarly help recover from user or administrator mistakes.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author.

Copyright is held by the owner/author(s).
SOSP'13, Nov. 3–6, 2013, Farmington, Pennsylvania, USA.
ACM 978-1-4503-2388-8/13/11.
http://dx.doi.org/10.1145/2517349.2522725

Aire's contribution over past work [7, 16] is in addressing three main challenges faced by intrusion recovery across web services:

Decentralized, asynchronous repair (§3). One possible design for a recovery system is to have a central repair coordinator that repairs all the services affected by an attack. However, this raises two issues. First, web services do not have a strict hierarchy of trust and so there is no single system that can be trusted to orchestrate repair across multiple services. Second, during repair, some services affected by an attack may be down, unreachable, or otherwise unavailable. Waiting for all services to be online in order to perform repair would be impractical and might unnecessarily delay recovery in services that are already online. Worse yet, an adversary might purposely add her own server to the list of services affected by an attack, in order to prevent timely recovery.

Aire solves these issues with two ideas. First, to avoid services having to trust a central repair coordinator, Aire performs *decentralized* repair: Aire defines a repair protocol that allows services to invoke repair on their past requests to other services, as well as their past responses to requests from other services. Second, to repair services after an intrusion without waiting for unavailable services, Aire performs *asynchronous* repair: a service repairs its local state as soon as it is asked to perform a repair, and if any past requests or responses are affected, it queues a repair message for other services, which can be processed when those services become available.

While Aire's repair infrastructure takes care of many issues raised by intrusion recovery, there are two remaining challenges that require application-specific changes to support repair:

Repair access control (§4). Repair operations themselves can be a security vulnerability, and an application must ensure that Aire's repair protocol does not give attackers new ways to subvert a web service. To this end, Aire provides an interface for applications to *specify access control policies* for every repair invocation.

Reasoning about partially repaired states (§5). With Aire's asynchronous repair, some services affected by an attack could be already repaired, while others might not have received or processed their repair messages yet. Such a partially repaired state could appear inconsistent to clients or other services, and lead to unexpected application behavior. To help developers handle partially repaired states in their applications, we propose the following contract: *repair should be indistinguishable from concurrent requests* issued by some client on the present state of the system. This contract largely reduces the problem of dealing with partially repaired states to the existing problem of dealing with concurrent clients, which many web application developers already have to reason about.

To evaluate Aire's design, we implemented a prototype of Aire for Django-based web applications. We ported three existing web applications to Aire: an open-source clone of StackOverflow called Askbot [1], a Pastebin-like application called Dpaste, and a Django-based OAuth service. We also developed our own shared spreadsheet application. In all cases, Aire required minimal changes to the application source code.

As there are no known attacks that propagate through interconnected web services in the wild, we construct four realistic intrusions that involve the above web applications, including a scenario inspired by the recent Facebook OAuth vulnerability, and demonstrate that Aire can recover from them. We also show that Aire can recover a subset of services from attack even when others are unavailable. Porting an application to Aire required changing under 100 lines of server-side code for the applications mentioned above. Supporting partial repair can require changing the API of a service; the most common example that we found is adding branches to a versioning API. Finally, we show that Aire's performance costs are moderate, amounting to a 19–30% CPU overhead and 6–9 KB/request storage cost in the case of Askbot.

2 Overview

Aire's goal is to undo the effects of an unwanted operation (specified by some user or administrator) that propagated through Aire-enabled services, which means producing a state that is consistent with the attack never having taken place. Aire expects that the user or administrator will pinpoint the unwanted operation (e.g., the initial intrusion into the system) to initiate recovery. In practice, the user or administrator will probably use some combination of auditing, intrusion detection, and analysis [17, 18] to find the initial intrusion point.

Aire assumes that each service exposes an API which defines a set of operations that can be performed on it, and that services and clients interact only via these operations; they cannot directly access each other's internal state. This model is commonplace in today's web services, such as Amazon S3, Facebook, Google Docs, and Dropbox. Under this model, an attack is an API operation that exploits a vulnerability or misconfiguration in a service and causes undesirable changes to the service's state. These state changes can propagate to other services, either as a result of this service invoking operations on other services or vice-versa. Aire aims to undo both the initial changes to the service state, as well as any changes propagated to other services.

On each individual service, Aire repairs the local state in a manner similar to the Warp intrusion recovery system [7], by rolling back the database state affected by the attack and re-executing past requests to the service that were affected by the attack. If during local repair Aire determines that requests or responses to other services might have been affected, Aire asynchronously sends repair messages to those services. Each repair message specifies which request or response was affected by repair, and includes a corrected version of the request or response. When a service receives a repair message, Aire initiates local recovery, after checking permissions for the repair message. Once repair messages propagate to all affected services, the attack's effects will be removed from the entire system. However, even before repair messages propagate everywhere, applications that are already online can repair their local state.

With API-level repair, Aire can recover from attacks that exploit misconfigurations of a service or vulnerabilities in a service's code, and from accidental user mistakes. This includes several scenarios, such as the ones described in §1, but does not include attacks on the OS kernel or the Aire runtime itself.

In the rest of this section we review Warp and present Aire's system architecture and assumptions.

2.1 Review of Warp

Aire's recovery of a service's local state is inspired by Warp [7]. This section provides a brief summary of Warp's design, and its rollback-redo approach to recovery, as it relates to Aire. Much as in Aire, an administrator of a Warp system initiates recovery by specifying an attack request to undo.

During normal execution of a web application, Warp builds up a repair log that will be used to recover from attacks. In particular, Warp records HTTP requests and their responses, and database queries issued by each request and their results. Warp also maintains a versioned database that stores all updates to every database row.

Given the above recorded information, Warp recovers from an attack as follows. First, Warp rolls back the database rows modified by the attack request to the request's original execution time. Second, Warp uses its logs to identify database queries that might have read the rows affected by the attack, or queries that might have modified the rows that have been rolled back, and re-executes the corresponding requests (except for the attack request, which is skipped). For each re-executed request, Warp rolls back the database rows accessed by that request to the time of the request's original execution, and applies the same algorithm to find other requests that might have been indirectly affected. This algorithm finishes after it has re-executed all requests affected by the attack, thereby reverting all of the attack's effects.

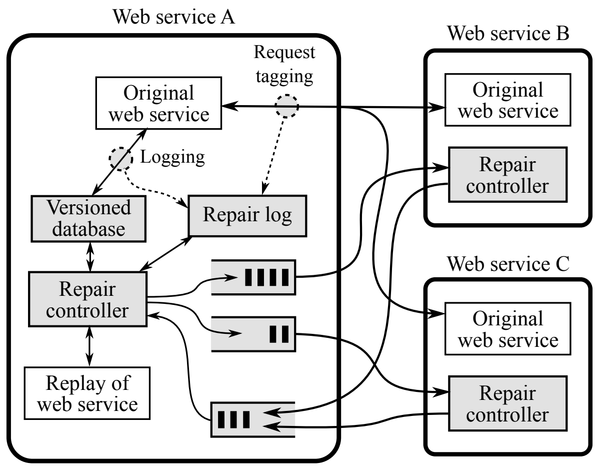

Figure 1: Overview of Aire's design. Components introduced or modified by Aire are shaded. Circles indicate places where Aire intercepts requests from the original web service. Not shown are the detailed components for services B and C.

2.2 Aire architecture

Figure 1 provides an overview of Aire's overall design. Every web service that supports repair through Aire runs an Aire repair controller, whose design is inspired by Warp [7]. The repair controller maintains a repair log during normal operation by intercepting the original service's requests, responses, and database accesses. The repair controller also performs repair operations as requested by users, administrators, or other web services, by rolling back affected state and re-executing affected requests.

In order to be able to repair interactions between services, Aire intercepts all HTTP requests and responses to and from the local system. Repairing requests or responses later on requires being able to name them; to this end, Aire assigns an identifier to every request and response, and includes that identifier in an HTTP header. The remote system, if it is running Aire, records this identifier for future use if it needs to repair the corresponding request or response.

During repair, if Aire determines that the local system sent an incorrect request or response to another service, it computes the correct request or response, and sends it along with the corresponding ID to the other service. Aire's repair messages are implemented as just another API on top of HTTP (with one special case, when a server needs to get in touch with a past client). Aire supports four kinds of operations in its repair API. The two most common repair operations involve replacing either a request or response with a different payload. Two other operations arise when Aire determines that the local service should never have issued a request in the first place, or that it should have issued a request while none was originally performed; in these cases, Aire asks the remote service to either cancel a past request altogether, or to create a new request.

Command and parameters	Description
`replace` (*request_id*, *new_request*)	Replaces past request with new data
`delete` (*request_id*)	Deletes past request
`create` (*request_data*, *before_id*, *after_id*)	Executes new request in the past
`replace_response` (*response_id*, *new_response*)	Replaces past response with new data

Table 1: The repair protocol between Aire servers.

When repairing a request, Aire updates its repair log and versioned database, just like it does during normal operation, so that a future repair can perform recovery on an already repaired request. This is important because asynchronous repair can cause a request to be repaired several times as repair propagates through all the affected services.

Aire must control who can issue repair operations, to ensure that clients or other web services cannot make unauthorized changes via the repair interface. Aire delegates this access control decision to the original service, as access control policies can be service-specific: for example, a service might require a stronger form of authentication (e.g., Google's two-step authentication) when a client issues a repair operation than when it issues a normal operation; or a platform such as Facebook might block repair requests from a third-party application if the application is attempting to modify the profile of a user that has since uninstalled that application.

In some cases, appropriate credentials for issuing a repair operation on another web service may be unavailable. For example, Aire on a service *A* may need to repair a request it previously issued on behalf of a specific user to a remote service *B*; however, if *A* no longer has the user's credentials to *B*, it cannot invoke repair on *B*. Aire treats this situation as if service *B* is not available, and queues the repair on *A* for later. Once the user logs in to *A* and provides credentials for *B*, *A* can use the user's credentials to propagate repair.

2.3 Assumptions

To perform repair, Aire makes several assumptions.

First, Aire assumes that each service's web software stack is in the trusted computing base. This includes the OS, the language runtime, the database server, and the application framework (such as Rails or Django) that the service operates on. Aire cannot recover from an attack that compromises these system components.

Second, Aire's repair propagation assumes that the services and clients affected by an attack are running Aire. If some client or service does not run Aire, then Aire will not be able to repair the effects of the attack on that client or service (and any other clients and services to which the attack spread from there). If a service cannot propagate repair to another machine, Aire notifies the service's administrator of the repair that cannot be propagated, so that the administrator can take remedial action (e.g., manual recovery).

As a corollary, Aire assumes that attacks do not propagate through Web browsers, as our current Aire prototype does not support browser clients, and hence cannot track or repair from attacks that spread through users' browsers. It may be possible to add repair for browsers in a manner similar to Warp's shadow browser [7].

Finally, Aire assumes that each service has an appropriate access control policy that denies access to unauthorized clients requesting repair, and that each service and its clients support partially repaired states. If the former assumption does not hold, attackers would be able to use repair to make unauthorized changes to a service. If the latter assumption is broken, clients may behave incorrectly due to inconsistencies between services.

3 Distributed repair

The next three sections delve into the details of Aire's design. This section describes Aire's asynchronous repair protocol, §4 focuses on permission checking for repair messages between services, and §5 discusses how applications can handle partially repaired states that arise during asynchronous repair.

3.1 Repair protocol

Each Aire-enabled web service exports a repair interface that its clients (including other Aire-enabled web services) can use to initiate repair on it. Aire's repair interface is summarized in Table 1.

Aire's repair begins when some client (either a user or administrator, or another web service) determines that there was a problem with a past request, or that it incorrectly missed issuing a past request. The client initiates repair on the corresponding service by using the `replace` or `delete` operations to fix the past request, or by using the `create` operation to create a new request in the past. Sometimes, a past response of a service is incorrect, in which case the service initiates repair on the corresponding client using the `replace_response` operation. We now describe these operations in more detail.

Repairing previous requests. The simplest operation is `replace`, which allows a client to indicate that a past request (named by its *request_id*) was incorrect, and should be replaced with *new_request* instead. The new request contains the corrected version of the arguments that were originally provided to the original request, including the URL, HTTP headers, query parameters, etc. When Aire's controller performs a `replace` operation, it repairs the local state to be as if the newly supplied request happened instead of the original request. If other requests or responses turn out to be affected by this repair, Aire queues appropriate repair API calls for other services.

The `delete` operation is similar to `replace`, but it is used when a client determines that it should not have issued some request at all. In this case, `delete` instructs the Aire repair controller to eliminate all side-effects of the request named by *request_id*.

Creating new requests. Sometimes, repair requires adding a new request "in the past." For example, if an administrator forgot to remove a user from an access control list when he should have been removed, one way to recover from this mistake is to add a new request at the right time in the past to remove the user from the access control list. The `create` call allows for this scenario.

One challenge with `create` is in specifying the time at which the request should execute. Different web services do not share a global timeline, so the client cannot specify a single timestamp that is meaningful to both the client and the service. Instead, the client specifies the time for the newly created request relative to other messages it exchanged with the service in the past. To do this, the client first identifies the local timestamp at which it wishes the created request to execute; then it identifies its last request before this timestamp and the first request after this timestamp that it exchanged with the service, and instructs the service to run the created request at a time between these two requests. The *before_id* and *after_id* parameters to the `create` call name these two requests.

The above scheme is not complete: it allows the client to specify the order of the new request with respect to past requests the client exchanged with the service executing the new request, but it does not allow the client to specify ordering with respect to arbitrary messages in the system. More general ordering constraints would require services to exchange large vector timestamps or dependency chains, which could be costly. As we have not yet found a need for it, we have not incorporated it into Aire's design.

Repairing responses. The `replace_response` operation allows a server to indicate that a past response to a client, named by its *response_id*, was incorrect, and to supply a corrected version of the response in *new_response*.

In web services, clients initiate communication to the server. However, to invoke a `replace_response` on a client, the service needs to initiate communication to the client. This raises two issues. First, the server needs to know where to send the `replace_response` call for a client. To address this issue, Aire associates a *notifier URL* with each request; if the server wants to contact the client to repair the response, it sends a request to the associated notifier URL.

Second, once a client gets a `replace_response` call from a service, it needs to authenticate the service. During normal operation, as the client initiates communication, it typically authenticates the server by communicating with it over TLS (which verifies the server's X.509 certificate). To allow the client to use the same authentication mechanism during repair, the service sends only a *response repair token* to the client's notifier URL, instead of the entire `replace_response` call; when a client receives a response repair token, it contacts the server and asks the server to provide the `replace_response` call for that token. This way, the client can appropriately authenticate the server, by validating its X.509 certificate.

Integrating Aire with HTTP. In order to name requests and responses during subsequent repair operations, Aire must assign a name to every one of them. To do this, Aire interposes on all HTTP requests and responses during normal operation, and adds headers specifying a unique identifier that will be used to name every request.

To ensure these identifiers uniquely name a request (or response) on a particular server, Aire assigns the identifier on the service handling the request (or receiving the response); it becomes the responsibility of the other party to remember this identifier for future repair operations. Specifically, Aire adds an `Aire-Response-Id:` header to every HTTP request issued *from* a web service; this identifier will name the corresponding response. The server receiving this request will store the response identifier, and will use it later if the response must be repaired. Conversely, Aire adds an `Aire-Request-Id:` header to every HTTP response produced by a web service; this identifier assigns a name to the HTTP request that triggered this response. A client can use this identifier to refer to the corresponding request during subsequent repair. Aire also adds an `Aire-Notifier-URL:` header to every issued request.

To make it easier for clients to use Aire's repair interface, Aire's repair API encodes the request being repaired (e.g., *new_request* for `replace`) in the same way as the web service would normally encode this request. The type of repair operation being performed (e.g., `replace` or `delete`) is sent in an `Aire-Repair:` HTTP header, and the *request_id* being repaired is sent in an `Aire-Request-Id:` header. Thus, to fix a previous request, the client simply issues the corrected version of the request as it normally would, and adds the `Aire-Repair: replace` and `Aire-Request-Id:` headers to indicate that this request should replace a past operation. In addition to requiring relatively few changes to client code, this also avoids introducing infrastructure changes (e.g., modifying firewall rules to expose a new service).

3.2 Local repair

As part of local repair of a service, Aire re-executes API operations that were affected by the attack. It is possible that one of these operations will execute differently due to repair, and issue a new HTTP request that it did not issue during the original execution. In that case, Aire must issue a `create` repair call to the corresponding web service, in order to create a new request "in the past." Re-execution can also cause the arguments of a previously issued request to change, in which case Aire queues a `replace` message to the remote web service in question. One difficulty with both `create` and `replace` calls is that to complete local repair, the application needs a response to the HTTP requests in these calls. However, Aire cannot block local repair waiting for the response.

To resolve this tension, Aire tentatively returns a "time-out" response to the application's request, which any application must already be prepared to deal with; this allows local repair to proceed. Once the remote web service processes the `create` or `replace` operation, it will send back a `replace_response` that replaces the time-out response with the actual response to the application's request. At this point, Aire will perform another repair to fix up the response.

When re-execution skips a previously issued request altogether, Aire queues a `delete` message. Finally, if re-execution changes the response of a previously executed request, or computes the response for a newly created request, Aire queues a `replace_response` message.

Aire maintains an outgoing queue of repair messages for each remote web service. If multiple repair messages refer to the same request or the same response, Aire can collapse them, by keeping only the most recent repair message. Sometimes, Aire might be unable to send a repair message, either because the original request or response did not include the dependency-tracking HTTP headers identifying the web service to send the message to, or because the communication to the remote web service timed out; in either case, Aire notifies the application (as we discuss in §4). Aire also aggregates incoming repair messages in an incoming queue, and can apply the changes requested by multiple repair operations as part of a single local repair.

3.3 Convergence

Recall that Aire's goal is to produce a state that is consistent with the attack never having taken place (*attack-free* for short). We will now informally argue that Aire's repair protocol converges to this state, assuming no failures (e.g., unreachable services or insufficient credentials, which are discussed in the next section).

Consider the list L of all messages (requests and responses between services) that were affected by the attack, sorted by receive time. We will argue that Aire eventually repairs the recipients of all these messages (i.e., servers that ran affected requests or clients that received affected responses). For simplicity, assume that each service keeps a complete timeline of its state (e.g., a checkpoint at every point in time), and that each service handled a request or response instantaneously at the time it was received. As repair messages propagate between services, the state timeline of each service will be repaired up to increasingly more recent points in time, eventually reaching the present.

The first message in L must be the initial attack at time t_0. Local repair rolls back any state modified as a result of this message to before t_0, and possibly re-executes some operations on that service. Since inputs to the service up to and including t_0 are now attack-free, the state timeline of that service is now attack-free up to and including t_0. All other services are also attack-free up to and including t_0, since the attack did not propagate to any other services as of t_0.

Now we argue by induction on the times at which messages in L were received. Suppose that all state timelines are attack-free as of some t_i, and the next message m in L is at $t_{i+1} > t_i$. Consider the service s that sent m as a result of some execution e. We know that m was sent at or before t_{i+1}, that s received no attack-affected message between t_i and t_{i+1}, and that the timeline of s is attack-free up to and including t_i. This means that all inputs to e up to the point when it sent m are now attack-free. Thus, local repair on s will re-execute e, produce the attack-free version of m, and send a repair message for m. Once the recipient of this repair message performs local repair, its timeline (and the timelines of all other services) will be attack-free up to and including t_{i+1}. By induction, Aire will eventually repair all timelines to the present.

In addition to producing the goal state, we would like Aire to eventually stop sending repair messages. This is true as long as the local repair implementation is *stable*: that is, when processing a repair message for time t, it produces repair messages only for requests or responses at times after t (i.e., does not change its mind about previous messages). Stable local repair ensures that repair messages progress forward in time starting from the attack, and eventually converge upon reaching the present time. Local repair is stable if re-execution is deterministic, which Aire achieves by recording and replaying sources of non-determinism as in Warp [7].

4 Repair access control

Access control is important because Aire's repair itself must not enable new ways for adversaries to propagate from one compromised web service to another. For ex-

Function and parameters	Description
Functions implemented by the web service and invoked by Aire	
`authorize` (*repair_type*, *original*, *repaired*)	Checks if a repair message should be allowed
`notify` (*msg_id*, *repair_type*, *original*, *repaired*, *error*)	Notifies application of a problem with a remote repair message
Functions implemented by Aire and invoked by the web service	
`retry` (*msg_id*, *updated_repair_type*, *updated_message*)	Resends a repair message

Table 2: The interface between Aire and the web service.

ample, a hypothetical design that allows any client with a valid request identifier to issue repair calls for that request is unsuitable, because an adversary that compromises a service storing many past request identifiers would be able to make arbitrary changes to those past requests, affecting many other services; this is something an attacker would not be able to do in the absence of Aire.

Aire requires that every repair API call be accompanied with credentials to authorize the repair operation. Aire delegates access control decisions to the application because principal types, format of credentials, and access control policies can be application-specific. For example, some applications may use cookies for authentication while others may include an access token as an additional HTTP header; and some applications may allow any user with a currently valid account to repair a past request issued by that user, while others may allow only users with special privileges to invoke repair. In the special case of `replace_response` messages, the repair message can be authenticated using the server's X.509 certificate, as discussed in §3.1, although an application developer can require (and supply) other credentials if needed.

The interface between Aire and an Aire-enabled service is shown in Table 2. Services running Aire export an `authorize` function that Aire invokes when it receives a repair message; Aire passes to the function the type of repair operation (`create`, `replace`, `delete`, or `replace_response`), and the original and new versions of the request or response to repair (denoted by the *original* and *repaired* parameters). The `authorize` function's return value indicates whether the repair should be allowed (using credentials from the *repaired* message); if the repair is not allowed, Aire returns an authorization error for the repair message.

To perform access control checks, the service may need to read old database state (e.g., to look up the principal that issued the original request). For this purpose, Aire provides the application read-only access to a snapshot of Aire's versioned database at the time when the original request executed; the specific interface depends on how the application's web framework provides database access (e.g., Django's ORM). Once a repair operation is authorized, Aire re-executes the new request, if any. As part of request re-execution, the application can apply other authorization checks, in the same way that it does for any other request during normal operation.

If a repair message sent to a remote server returns an authorization error (e.g., because the credentials have expired) or times out, Aire notifies the application of the error by invoking the `notify` function (and continues to process other repairs in the meantime). Once the application obtains appropriate credentials for a failed repair operation, it can use the `retry` function to ask Aire to resend the repair message. In the OAuth example above, the client application could display the failed repair message to the user whose OAuth token was stale, and prompt the user for a fresh OAuth token or ask if the message should be dropped altogether.

5 Reasoning about partially repaired state

Aire's asynchronous repair exposes the state of a service to its clients immediately after local repair is done, without waiting for repair to complete on other affected services. In principle, for a distributed system composed of arbitrary tightly coupled services, a partially repaired state can appear invalid to clients of the services. For example, if one of the services is a lock service, and during repair it grants a lock to a different application than it did during original execution, then in some partially repaired state both the applications could be holding the lock; this violates the service's invariant that only one application can hold a lock at any time, and can confuse applications that observe this partially repaired state.

However, Aire is targeted at web services, which are loosely coupled, in part because they are under different administrative domains and cannot rely on each other to be always available. In practice, for such loosely coupled web service APIs, exposing partially repaired states does not violate their invariants. In the rest of this section, we first present a model to reason about partially repaired states, and then provide an example of how a developer can modify a service to handle partially repaired states if necessary.

5.1 Modeling repair as API invocations

Many web services and their clients are designed to deal with concurrent operations, and so web application developers already have to reason about concurrent updates. For example, Amazon S3, a popular web service offering

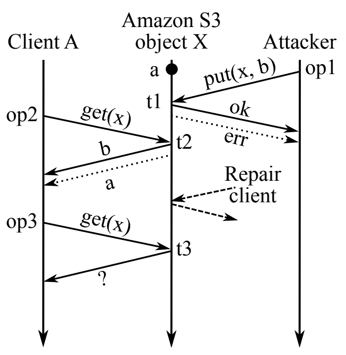

Figure 2: Example scenario demonstrating modeling repair actions as concurrent operations by a repair client. Solid arrows indicate requests during original execution, and dotted arrows indicate eventual repair propagation. The S3 service initiates local repair in between times $t2$ and $t3$ by deleting the attacker's `put`. If S3's local repair completes before $t3$, op3 observes value a for X. If A has not yet received the propagated repair from S3, receiving the value a for X at time $t3$ is equivalent to a concurrent writer (the hypothetical repair client) doing a concurrent `put(x, a)`.

a data storage interface, supports both a simple PUT/GET interface that provides last-writer-wins semantics in the face of concurrency, and an object versioning API that helps clients deal with concurrent writes.

Building on this observation, we propose a contract: any repair of a service should be indistinguishable from a hypothetical *repair client* performing normal API calls to the service, in the present time. Note that this is just a way of reasoning about what effects a repair can have; Aire's repair algorithm does not actually construct such a sequence of API calls. If repair operations are equivalent to a concurrent client, then application developers can handle partially repaired states simply by reasoning about this additional concurrent client, rather than having to reason about all possible timelines in which concurrent repair operations are happening. In particular, applications that already handle arbitrary concurrent clients require no changes to properly handle partially repaired states.

This model fits many existing web services. For example, consider the scenario in Figure 2, illustrating operations on object X stored in Amazon S3. Initially, X had the value a. At time $t1$, an attacker writes the value b to X. At time $t2$, client A reads the value of X and gets back b. At time $t3$ the client reads the value of X again. In the absence of repair or any concurrent operations, A should receive the value b, but what should happen if, between $t2$ and $t3$, Amazon S3 determines the attacker's write was unauthorized and `delete`s that request?

If repair occurs between $t2$ and $t3$, the state of X will roll back to a, and two things will happen with A: first, it will receive a in response to its second request at $t3$, and second, at some later time it will receive a `replace_response` from S3 that provides the repaired response for the first `get`, also containing a. Client A observes partially repaired state during the time between when local repair on S3 completed (which is sometime between $t2$ and $t3$) and when A finally receives the `replace_response` message; A sees this state as valid (with its first `get(x)` returning b and its second `get(x)` returning a), because a hypothetical repair client could have issued a `put(x, a)` in the meantime.

5.2 Making service APIs repairable

A web service with many concurrent clients that offers only a simple PUT/GET interface can handle partially repaired states, because clients cannot make any assumptions about the state of the service in the face of concurrency. However, some web service APIs provide stronger invariants that require application changes to properly handle partial repair. For example, some web services provide a versioning API that guarantees an immutable history of versions (as we discuss in §7.3). Suppose client A from our earlier example in Figure 2 asked the server for a set of all versions of X, instead of a `get` on the latest version. At time $t2$, A would receive the set of versions $\{a, b\}$. If repair simply rolled back the state of X between $t2$ and $t3$, A would receive the set of versions $\{a\}$ at time $t3$ with b removed from the set, a state that no concurrent writer could have produced using the versioning API. The rest of this section describes how a developer can modify an application to handle partially repaired states, using a versioning interface as an example.

Consider a web service API that provides a single, linear history of versions for an object. Once a client performs a `put(x, b)`, the value b must appear in the history of values of x (until old versions are garbage-collected). If the `put(x, b)` was erroneous and needs to be `delete`d, what partially repaired state can the service expose? Removing b from the version history altogether would be inconsistent if the service does not provide any API to delete old versions, and might confuse clients that rely on past versions to be immutable. On the other hand, appending new versions to the history (i.e., writing a new fixed-up value) prevents Aire from repairing past responses. In particular, if a past request asked for a set of versions, Aire would have to send a new set of versions to that client (using `replace_response`) where the effects of b have been removed. However, if Aire extends that past version history by appending a new version that reverts b, this synthesized history would be inconsistent with the present history.

One way to handle partial repair with a versioning API is to extend the API to support branches, similar to the model used by git [6]. With an API that supports branches, when a past request needs to be repaired, Aire can create a new branch that contains a repaired set of changes, and move the "current" pointer to the new branch, while preserving the original branch. This allows the API to handle partially repaired states, and has the added benefit

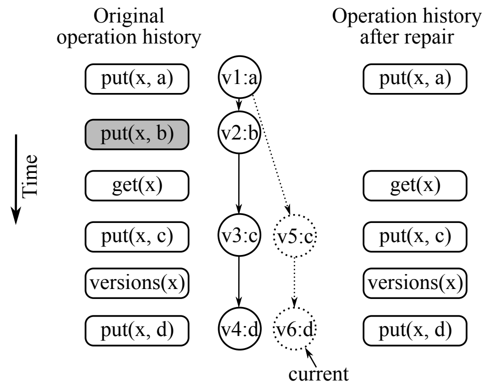

Figure 3: Repair of a single key in a versioned key-value store. Repair starts when the shaded operation `put(x, b)` from the original history, shown on the left, is `deleted`. This leads to the repaired history of operations shown on the right. The version history exposed by the API is shown in the middle, with two branches: the original chain of versions, shown with solid lines, and the repaired chain of versions, dotted. In each immutable version `v:w`, `v` is the version number and `w` is the value of the key. The mutable "current" pointer moves from one branch to another as part of repair.

of preserving the history of all operations that happened, including mistakes or attacks, instead of erasing them.

For example, consider a simple key-value store that maintains a history of all values for each key, as illustrated in Figure 3. In addition to `put` and `get` calls, the key-value store provides a `versions(x)` call that returns all previous versions of key x. During repair, request `put(x, b)` is `deleted`. An API with linear history does not allow clients to handle partial repair (as discussed above), but a branching API does. With branches, repair creates a new branch (shown in the right half of Figure 3), and re-applies legitimate changes to that branch, such as `put(x, c)`. These changes will create new versions on the new branch, such as $v5$ mirroring the original $v3$ (the application must ensure that version numbers themselves are opaque identifiers, even though we use sequential numbers in the figure). At the end of local repair, Aire exposes the repaired state, with the "current" branch pointer moved to the repaired branch. This change is consistent with concurrent operations performed through the regular web service API.

For requests whose responses changed due to repair, Aire sends `replace_response` messages that contain the new responses: for a `get` request, the new response is the repaired value at the logical execution time of the request, and for a `versions` request, it contains only the versions created before the logical execution time of the request. In the example of Figure 3, the new response for `get(x)` is a (replacing b), while the new response for the `versions(x)` call is $\{v1, v2, v3, v5\}$ (replacing $\{v1, v2, v3\}$), and does not contain $v4$ and $v6$.

6 Implementation

We implemented a prototype of Aire for the Django web application framework [2]. Aire leverages Django's HTTP request processing layer and its object-relational mapper (ORM). The ORM abstracts data stored in an application's database as Python classes (called "models") and relations between them; an instance of a model is called a model object.

We modified the Django HTTP request processor and the Python `httplib` library functions to intercept incoming and outgoing HTTP requests, assign IDs to them, and record them in the repair log. To implement versioning of model objects, we modified the Django ORM to intercept the application's reads and writes to model objects. On a write, Aire transparently creates a new version of the object, and on a read, it fetches the latest version during normal execution and the correct past version during local repair. Aire rolls back a model object to time t by deleting all versions after t. In addition to tracking dependencies between writes and reads to the same model, Aire also tracks dependencies between models (such as unique key and foreign key relationships) and uses them to propagate repair. We modified about 3,000 lines of code in Django to implement the Aire interceptors; the Aire repair controller was another 2,800 lines of Python code.

Repair for a versioned API. If a service implements versioning, it indicates this to Aire by making the model class for its immutable versions a subclass of Aire's `AppVersionedModel` class. `AppVersionedModel` objects are not rolled back during repair, and Aire does not perform versioning for these objects. If other model objects store references to `AppVersionedModel` objects, Aire rolls those other objects back during repair.

7 Application case studies

This section answers the following questions:

- What kinds of attacks can Aire recover from?
- How much of the system is repaired if some services are offline or authorization fails at a service?
- How much effort is required to start using Aire in an existing application?

7.1 Intrusion recovery

As we do not know of any significant compromises that propagated through interconnected web services to date, to evaluate the kinds of attacks that Aire can handle, we implemented four attack scenarios and attempted to recover from each attack using Aire. The rest of this subsection describes these scenarios and how Aire handled the attacks.

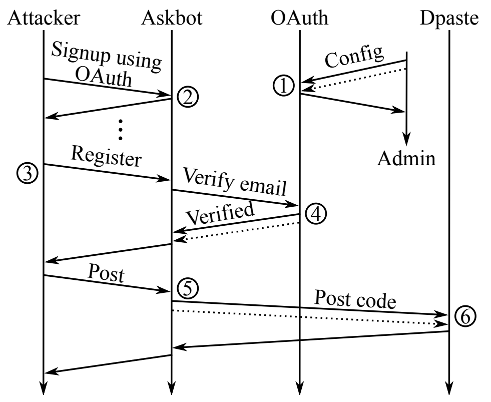

Figure 4: Attack scenario in Askbot demonstrating Aire's repair capabilities. Solid arrows show the requests and responses during normal execution; dotted arrows show the Aire repair operations invoked during recovery. Request ① is the configuration request that created a vulnerability in the OAuth service, and the attacker's exploit of the vulnerability results in requests ②-⑥. For clarity, requests in the OAuth handshake, other than request ②, have been omitted.

Askbot. A common pattern of integration between web services is to use OAuth or OpenID providers like Facebook, Google, or Yahoo, to authenticate users. If an attacker compromises the provider, she can spread an attack to services that depend on the provider. To demonstrate that Aire can recover from such attacks, we evaluated Aire using real web applications, with an attack that exploits a vulnerability similar to the ones recently discovered in Facebook [9, 10].

The system for the scenario consists of three large open-source Django web services: Askbot [1], which is an open-source question and answer forum similar to Stack Overflow and used by sites like the Fedora Project; Dpaste, a Django-based pastebin service, which allows posting and sharing of code snippets; and a Django-based OAuth service. These three services together comprise 183,000 lines of Python code, excluding blank lines and comments. Askbot maintains a significant amount of state, including questions and answers, tags, user profiles, ratings, and so on, which Aire must repair. We modified Askbot to integrate with Django OAuth and Dpaste, which it did not do out-of-the-box; these modifications took 74 and 27 lines of Python code, respectively.

The attack scenario is shown in Figure 4. We configured Askbot to allow users to sign up using accounts from an external OAuth provider service that we set up for this purpose. A user's signup in our Askbot setup is depicted by the requests ②-④ in Figure 4. As is typical in an OAuth handshake, Askbot redirects the user to the OAuth provider service. The user logs in to the OAuth service, and the OAuth service grants an OAuth token to Askbot if the user allows it to do so. To keep Figure 4 simple, only the first request in the OAuth handshake (request ②) is shown. Once Askbot gets an OAuth token for the user, it allows the user to register with an email address (request ③) that it verifies with the OAuth service using the user's OAuth token (request ④). If the verification succeeds, Askbot creates a local account for the user and allows the user to post and view questions and answers.

In addition to OAuth integration, we also modified Askbot to integrate with Dpaste; if a user's Askbot post contains a code snippet, Askbot posts this code to the Dpaste service for easy viewing and downloading by other users. Finally, the Askbot service also sends users a daily email summarizing that day's activity. These loosely coupled dependencies between the services mimic the dependencies that real web services have on each other.

The attack we simulate in this scenario is based on a recent Facebook vulnerability [9]. To enable the attack, we added a debug configuration option in the OAuth service that always allows email verification to succeed (adding this option required modifying 13 lines of Python code in the OAuth service). This option is mistakenly turned on in production by the administrator by issuing request ①, thus exposing the vulnerability. The attacker exploits this vulnerability in the OAuth service to sign up with Askbot as a victim user (requests ②-④) and post a question with some code (request ⑤), thereby spreading the attack from the OAuth service to Askbot. Askbot automatically posts this code snippet to Dpaste (request ⑥), spreading the attack further. Later, a legitimate user views and downloads this code from Dpaste, and at an even later time, Askbot sends a daily email summary containing the attacker's question, creating an external event that depends on the attack; both of these events are not shown in the figure. Before, after, and during the attack, other legitimate users continue to use the system, logging in, viewing and posting questions and answers, and downloading code from the Dpaste service. Some actions of these legitimate users, such as posting their own questions, are not dependent on the attack, while others, such as reading the attacker's question, are dependent.

We used Aire to recover from the attack. The administrator starts repair by invoking a `delete` operation on request ①, which introduced the vulnerability. The `delete` is shown by the dotted arrow corresponding to ① in Figure 4. This initiates local repair on the OAuth service, which deletes the misconfiguration, and invokes a `replace_response` operation on request ④ with an error value for the new response. The `replace_response` propagates repair to Askbot: as requests ③ and ⑤ depend on the response to request ④, local repair on Askbot re-executes them using the new error response, thereby undoing the attacker's signup (request ③) and the attacker's post (request ⑤). Local repair on Askbot also runs a compensating action for the daily summary email, which notifies the Askbot administrator of the new email contents without the attacker's question; it re-executes all legiti-

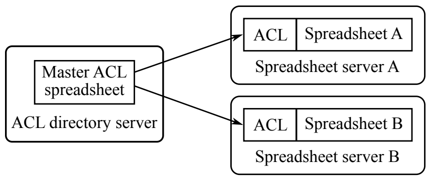

Figure 5: Setup for the spreadsheet application attack scenarios. Arrows represent the ACL directory server updating ACLs on other spreadsheet servers.

mate user requests that depended on the attack requests; and finally it invokes a `delete` operation on Dpaste to cancel request ⑥. Dpaste in turn performs local repair, resulting in the attacker's code being deleted, and a notification being sent to the user who downloaded the code. This completes recovery, which removes all the effects of the attack and does not change past legitimate actions in the system that were not affected by the attack.

Lax permissions. A common source of security vulnerabilities comes from setting improper permissions. In a distributed setting, we consider a scenario where one service maintains an authoritative copy of an access control list, and periodically updates permissions on other services based on this list, similar to the example presented in §1 to motivate the need for Aire. If a mistake is made in the master list, it is important not only to propagate a remedy to other services, but also to undo any requests that took advantage of the mistake on those services.

Since Askbot did not natively support such a permission model, we implemented our own spreadsheet service for this scenario. The spreadsheet service has a simple scripting capability similar to Google Apps Script [12]. This allows a user to attach a script to a set of cells, which executes when values in cells change. We use scripting to implement a simple distribution mechanism for access control lists (ACLs). The setup is shown in Figure 5. The *ACL directory* is a spreadsheet service that stores the master copy of the ACL for the other two spreadsheet services. A script on the directory updates the ACLs on the other services when an ACL on the directory is modified.

The attack is as follows: an administrator mistakenly adds an attacker to the master copy of the ACL by issuing a request to update the ACL directory; the ACL script distributes the new ACL to spreadsheets *A* and *B*. Later, the attacker takes advantage of these extra privileges to corrupt some cells in both spreadsheets. All this happens while legitimate users are also using the services.

Once the administrator realizes his mistake, he initiates repair by invoking a `delete` operation on the ACL directory to cancel his update to the ACL. The ACL directory reverts the update, and invokes `delete` on the two requests made by its script to distribute the corrupt ACL to the two services. This causes local repair on each of the two services, which rolls back the corrupt ACL. All the requests since the corrupt ACL's distribution are re-executed, as every request to the service checks the ACL. As the attacker is no longer in the ACL, her requests fail, whereas the requests of legitimate users succeed; Aire thereby cleans up the attacker's corrupt updates while preserving the updates made by legitimate users.

Lax permissions on the configuration server. A more complex form of the above attack could take place if the ACL directory itself is misconfigured. For example, suppose the administrator does not make any mistakes in the ACLs in the directory, but instead accidentally makes the directory world-writable. An adversary could then add herself to the master copy of the ACL for spreadsheets *A* and *B*, wait for updates to propagate to *A* and *B*, and then modify data in those spreadsheets as above.

Recovery in this case is more complicated, as it needs to revert the attacker's changes to the ACL directory in addition to the spreadsheet servers. Repair is initiated by the administrator invoking a `delete` operation on his request that configured the ACL directory to be world-writable. This initiates local repair on the ACL directory, reverting its permissions to what they were before, and cancels the attacker's request that updated the ACL. This triggers the rest of the repair as in the previous scenario, and fully undoes the attack.

Propagation of corrupt data. Another common pattern of integration between services is synchronization of data, such as notes and documents, between services. If an attack corrupts data on one service, it automatically spreads to the other services that synchronize with it.

To evaluate Aire's repair for synchronized services, we reused the spreadsheet application and the setup from the previous scenarios, and added synchronization of a set of cells from spreadsheet service *A* to spreadsheet service *B*. A script on *A* updates the cells on *B* whenever the cells on *A* are modified. As before, the attack is enabled by the administrator mistakenly adding the attacker to the ACL. However, the attacker now corrupts a cell only in service *A*, and the script on *A* automatically propagates the corruption to *B*.

Repair is initiated, as before, with a `delete` operation on the ACL directory. In addition to the repair steps performed in the previous scenario, after service *A* completes its local repair, it invokes a `delete` operation on service *B* to cancel the synchronization script's update of *B*'s cell. This reverts the updates made by the synchronization, thereby showing that Aire can track and repair attacks that spread via data synchronization as well.

7.2 Partial repair propagation

Repair might not propagate to all services if some services are offline during repair or a service rejects a repair message as unauthorized. To evaluate Aire's partial repair due to offline services, we re-ran the Askbot repair experiment with Dpaste offline during repair. Local repair runs on both the OAuth and Askbot services; the vulnerability in the OAuth service is fixed and the attacker's post to Askbot is deleted. Clients interacting with the OAuth and Askbot services see the state with the attacker's post deleted, which is a valid state, as this could have resulted due to a concurrent operation by another client. Most importantly, this partially repaired state immediately prevents any further attacks using that vulnerability, without having to wait for Dpaste to be online. Once Dpaste comes online, repair propagates to it and deletes the attacker's post on it as well. When we re-ran the experiment and never brought Dpaste back online, Aire on Askbot timed out attempting to send the `delete` message to Dpaste, and notified the Askbot administrator, so that he could take remedial action.

We also ran the spreadsheet experiments with service *B* offline. In all cases, this results in local repair on service *A*, which repairs the corrupted cells on *A*, and prevents further unauthorized access to *A*. Once *B* comes online and the directory server or *A* or both propagate repair to it (depending on the specific scenario), *B* is repaired as well. Similar to the offline scenario in Askbot, clients accessing the services at any time find the services' state to be valid; all repairs to the services are indistinguishable from concurrent updates.

Finally, we used the spreadsheet experiments to evaluate partial repair due to an authorization failure of a repair message. We use an OAuth-like scheme for spreadsheet services to authenticate each other—when a script in a spreadsheet service communicates with another service, it presents a token supplied by the user who created the script. The spreadsheet services implement an access control policy that allows repair of a past request only if the repair message has a valid token for the same user on whose behalf the request was originally issued.

We ran the spreadsheet experiments and initiated repair after the user tokens for service *B* have expired. This caused service *B* to reject any repair messages, and Aire effectively treats it as offline; this results in partially repaired state as in the offline experiment described before. On the next login of the user who created the script, the directory service or *A* (depending on the experiment) presents the user with the list of pending repair messages. If the user refreshes the token for service *B*, Aire propagates repair to *B*, repairing it as well.

The results of these three experiments demonstrate that Aire's partial repair can repair the subset of services that are online, and propagate repair once offline services or appropriate credentials become available.

Service	**Simple CRUD**	**Versioned**	**Description**
Amazon S3	✓	✓	Simple file storage
Google Docs	✓	✓	Office applications
Google Drive	✓	✓	File hosting
Dropbox	✓	✓	File hosting
Github	✓	✓	Project hosting
Facebook	✓		Social networking
Twitter	✓		Social microblogging
Flickr	✓		Photo sharing
Salesforce	✓		Web-based CRM
Heroku	✓		Cloud apps platform

Table 3: Kinds of interfaces provided by popular web service APIs to their clients.

7.3 Porting applications to use Aire

Porting an application to use Aire involves changes both on the client and the server side of that application's interface. This section explores the two in turn, demonstrating that Aire requires little changes to both client-side and server-side code to support repair.

Client-side porting effort. Clients of an Aire-enabled service must be prepared to deal with partially repaired states. To understand what would be involved for a client to handle such states, we examine the interfaces of 10 popular web services for operations that might expose inconsistencies as a result of partial repair. We found that the APIs fell into two categories, as shown in Table 3.

Every service offered a simple CRUD (create, read, update, delete) interface on the resource objects exported by the service. There is no concurrency control for the simple CRUD operations, and if multiple updates happen simultaneously, the last update wins. For such services, handling partial repair in a client boils down to assuming there is an additional repair client that can perform updates at any moment. In many situations, clients are already prepared to deal with concurrent updates from other clients, and thus would require little additional effort to support Aire.

Half of the services studied also provide a versioning API to deal with concurrent updates, typically exposing a linear history of immutable versions for each resource. These APIs allow clients to fetch a resource's version list, to perform an update only if a resource is at the right version, and to restore a resource to a past version (which creates a new version with the contents of the past version). In this case, the interface would have to be changed to support branching, as discussed in §5.2 (for services that don't already support it). The client would then be able to assume that existing versions on a branch are immutable, but that the current branch pointer could be switched by a concurrent repair client at any time.

Server-side porting effort. To evaluate the effort of adopting server-side application code to use Aire, we measure the lines of code we changed in the Askbot, Dpaste, and Django OAuth applications to support Aire as described in §7.1. These three applications comprise 183,000 lines of code in total, excluding comments and blank lines. To run these applications with Aire, we added an `authorize` function to implement a repair access control policy. The policy allows repair of a past request only if the repair message is issued on behalf of the same user who issued the past request. Implementing the `authorize` function took 55 lines of Python code.

For the spreadsheet application, we used the same access control policy and `authorize` implementation. We also implemented support for user-initiated retry of repair, using the `notify` and `retry` interface, which we used in our partial repair experiments (§7.2). Adding this capability to the spreadsheet application required 26 lines of code (for comparison, the entire spreadsheet application is 925 lines of code).

To evaluate the difficulty of implementing an Aire-compatible service with a versioning API, we again used our spreadsheet example application. We first implemented a simple linear versioning scheme, where each version is just an incrementing counter. We then extended it to support version trees so that clients could handle Aire's partial repair. This involved adding parent and timestamp fields to each version, and a pointer to the current version for each cell. This required modifying 44 lines of code.

8 Performance evaluation

To evaluate Aire's performance, this section answers the following questions:

- What is the overhead of Aire during normal operation, in terms of CPU overhead and disk space?
- How long does repair take on each service, and for the entire system?

We performed experiments on a server with a 2.80 GHz Intel Core i7-860 processor and 8 GB of RAM running Ubuntu 12.10. As our prototype's local repair is currently sequential, we used a single core with hyperthreading turned off to make it easier to reason about overhead.

8.1 Overhead during normal operation

To measure Aire's overhead during normal operation, we ran Askbot with and without Aire under two workloads: a write-heavy workload that creates new Askbot questions as fast as it can, and a read-heavy workload that repeatedly queries for the list of all the questions. During both workloads, the server experienced 100% CPU load.

Table 4 shows the throughput of Askbot in these experiments, and the size of Aire's logs. Aire incurs a CPU overhead of 19% and 30%, and a per-request storage overhead of 5.52 KB and 9.24 KB (or 8 GB and 12 GB per day) for the two workloads, respectively. One year of logs should fit in a 3 TB drive at this worst-case rate, allowing for recovery from attacks during that period.

Workload	Throughput		Log size per req.	
	No Aire	Aire	App.	DB
Reading	21.58 req/s	17.58 req/s	5.52 KB	0.00 KB
Writing	23.26 req/s	16.20 req/s	8.87 KB	0.37 KB

Table 4: Aire overheads for creating questions and reading a list of questions in Askbot. The left two columns show throughput without and with Aire. The right two columns show the per-request storage required for Aire's logs (compressed) and the database checkpoints.

	Askbot	OAuth	Dpaste
Repaired requests	105 / 2196	2 / 9	1 / 496
Repaired model ops	5444 / 88818	9 / 128	4 / 7937
Repair messages sent	1	1	0
Local repair time	84.06 sec	0.10 sec	3.91 sec
Normal exec. time	177.58 sec	0.01 sec	0.02 sec

Table 5: Aire repair performance. The first two rows show the number of repaired requests and model operations out of the total number of requests and model operations, respectively.

8.2 Repair performance

To evaluate Aire's repair performance, we used the Askbot attack scenario from §7.1. We constructed a workload with 100 legitimate users and one victim user. The attacker signs up as the victim and performs the attack; during this time, each legitimate user logs in, posts 5 questions, views the list of questions and logs out. Afterwards, we performed repair to recover from the attack.

The results of the experiment are shown in Table 5. The two requests repaired in the OAuth service are requests ① and ④ in Figure 4, and the one request repaired in Dpaste is request ⑥. The repair messages sent by OAuth and Askbot are the `replace_response` for request ④ and the `delete` for request ⑥, respectively. Askbot does not send `replace_response` for requests ③ and ⑤, as the attacker browser's requests do not include a `Aire-Notifier-URL:` header.

Local repair on Askbot re-executes only the requests affected by the attack (105 out of the 2196 total requests), which results in repair taking less than half the time taken for original execution. The attack affects so many requests because the attack question was posted at the beginning of the workload, so subsequent legitimate users' requests to view the questions page depended on the attack request that posted the question. These requests are re-executed when the attacker's request is canceled, and their repaired responses do not contain the attacker's question. Aire on Askbot does not send `replace_response` messages for these requests as the user's browsers did not include a `Aire-Notifier-URL:` header.

Repair takes longest on Askbot, and it is the last to finish local repair. In our unoptimized prototype, repair for each request is ~10× slower than normal execution. This is because the repair controller and the replayed web service are in separate processes and communicate with each other for every Django model operation; optimizing the communication by co-locating them in the same process should improve repair performance.

9 Discussion and limitations

Aire recovers the integrity of a system after an attack by undoing unauthorized writes, but it cannot undo damage resulting from unauthorized reads, such as an attack that leaked confidential information. However, Aire could be extended to help an administrator identify leaks, so he can take remedial action—for example, if the administrator marked confidential data for Aire, Aire could notify him of reads that returned confidential data only during original execution but not during repair.

Aire's repair log and database of versioned Django model objects grow in size over time, and eventually garbage collection of old versions becomes necessary. When the administrator of a service determines that logs prior to a particular date are no longer needed, Aire performs garbage collection by deleting repair logs and versions of database rows before that date. Once garbage collection is done, Aire cannot repair requests to the service prior to that date; if a client issues a repair operation on a request whose logs were garbage collected, Aire treats the service as permanently unavailable and notifies the client's administrator.

Our current prototype does not support simultaneous normal execution and repair. When repair is invoked on a service, Aire stops normal operation, switches the service into repair mode, completes local repair, and switches it back to normal operation. Our implementation could be extended to support simultaneous normal execution and repair similar to Warp's repair generations [7].

10 Related work

The two closest pieces of work to Aire are the Warp [7] and Dare [16] intrusion recovery systems. Warp focuses on recovery in a single web service and is the inspiration for Aire's local recovery. Aire additionally tracks attacks that spread across services and recovers from them, and defines a model for reasoning about partially repaired state. Dare performs intrusion recovery on a cluster of machines. However, Dare's repair is synchronous and assumes that all machines are in the same administrative domain; both of these design decisions are incompatible with web services, unlike Aire's asynchronous repair.

Some web services, like Google Docs and Dropbox, already allow a user to roll back their files and documents to a previous version. Aire provides a more powerful recovery mechanism that tracks the spread of an attack across services and undoes all effects of the attack while preserving subsequent legitimate changes.

After a compromise, Polygraph [19] recovers the non-corrupted state in a weakly consistent replication system by rolling back corrupted state. However, unlike Aire, it does not attempt to preserve the effects of legitimate actions, which can lead to significant data loss.

Heat-ray [8] considers the problem of attackers propagating between machines within a single administrative domain, and suggests ways to reduce trust between machines. On the other hand, Aire is focused on attackers spreading across web services that do not have a single administrator, and allows recovery from intrusions. Techniques such as Heat-ray's could be helpful in understanding and limiting the ability of an adversary to spread from one service to another.

Akkuş and Goel's system [5] uses taint tracking to analyze dependencies between HTTP requests and database elements, and uses administrator guidance to recover from data corruption. However, unlike Aire, it can recover only from accidental corruption and not attacks, and it cannot handle attacks that spread across services.

The user-guided recovery system of Simmonds et al. [21] recovers from violations of application-level invariants in a web service and uses compensating actions and user input to resolve these violations. However, it cannot recover from attacks or accidental data corruption.

Past work on distributed debugging [4] intercepts executions of unmodified applications and tracks dependencies for performance debugging, whereas Aire tracks dependencies for recovery.

11 Conclusion

This paper presented Aire, an intrusion recovery system for interconnected web services. Aire introduced three key techniques for distributed repair: (1) a repair protocol to propagate repair across services that span administrative domains, (2) an asynchronous approach to repair that allows each service to perform repair at its own pace without waiting for other services, and (3) a contract that helps developers reason about states resulting from asynchronous repair. We built a prototype of Aire for Django and demonstrated that porting existing applications to Aire requires little effort, that Aire can recover from four realistic attack scenarios, and that Aire's repair model is supported by typical web service APIs.

Acknowledgments

We thank Frans Kaashoek, Eddie Kohler, the anonymous reviewers, and our shepherd, Michael Walfish, for their help and feedback. This research was supported by the DARPA Clean-slate design of Resilient, Adaptive, Secure Hosts (CRASH) program under contract #N66001-10-2-4089, and by NSF award CNS-1053143.

References

[1] Askbot – create your Q&A forum. `http://www.askbot.com`.

[2] Django: the Web framework for perfectionists with deadlines. `http://www.djangoproject.com`.

[3] OAuth community site. `http://oauth.net`.

[4] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, Oct. 2003.

[5] İ. E. Akkuş and A. Goel. Data recovery for web applications. In *Proceedings of the 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, Chicago, IL, June–July 2010.

[6] S. Chacon. *Pro Git*. Apress, Aug. 2009.

[7] R. Chandra, T. Kim, M. Shah, N. Narula, and N. Zeldovich. Intrusion recovery for database-backed web applications. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 101–114, Cascais, Portugal, Oct. 2011.

[8] J. Dunagan, A. X. Zheng, and D. R. Simon. Heat-ray: Combating identity snowball attacks using machine learning, combinatorial optimization and attack graphs. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, Oct. 2009.

[9] N. Goldshlager. How I hacked Facebook OAuth to get full permission on any Facebook account. `http://www.nirgoldshlager.com/2013/02/how-i-hacked-facebook-oauth-to-get-full.html`, Feb. 2013.

[10] N. Goldshlager. How I hacked any Facebook account...again! `http://www.nirgoldshlager.com/2013/03/how-i-hacked-any-facebook-accountagain.html`, Mar. 2013.

[11] N. Goldshlager. How I hacked Instagram accounts. `http://www.breaksec.com/?p=6164`, May 2013.

[12] Google, Inc. Google apps script, 2013. `https://script.google.com`.

[13] E. Hammer-Lahav. OAuth security advisory: 2009.1. `http://oauth.net/advisories/2009-1/`, Apr. 2009.

[14] ifttt, Inc. Put the internet to work for you, 2013. `https://ifttt.com`.

[15] T. Kim, X. Wang, N. Zeldovich, and M. F. Kaashoek. Intrusion recovery using selective re-execution. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 89–104, Vancouver, Canada, Oct. 2010.

[16] T. Kim, R. Chandra, and N. Zeldovich. Recovering from intrusions in distributed systems with Dare. In *Proceedings of the 3rd Asia-Pacific Workshop on Systems*, Seoul, South Korea, July 2012.

[17] T. Kim, R. Chandra, and N. Zeldovich. Efficient patch-based auditing for web application vulnerabilities. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 193–206, Hollywood, CA, Oct. 2012.

[18] S. T. King and P. M. Chen. Backtracking intrusions. *ACM Transactions on Computer Systems*, 23(1):51–76, Feb. 2005.

[19] P. Mahajan, R. Kotla, C. C. Marshall, V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, and T. Wobber. Effective and efficient compromise recovery for weakly consistent replication. In *Proceedings of the ACM EuroSys Conference*, Nuremberg, Germany, Mar. 2009.

[20] M. Mimoso. Twitter OAuth API keys leaked. `http://threatpost.com/twitter-oauth-api-keys-leaked-030713`, Mar. 2013.

[21] J. Simmonds, S. Ben-David, and M. Chechik. Guided recovery for web service applications. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Santa Fe, NM, Nov. 2010.

[22] Yahoo, Inc. Pipes: Rewire the web, 2013. `http://pipes.yahoo.com`.

[23] Zapier, Inc. Automate the web, 2013. `https://zapier.com`.