MIT-CSAIL-TR-2007-041 August 6, 2007

Toward Secure Services from Untrusted Developers

Micah Brodsky, Petros Efstathopoulos, Frans Kaashoek, Eddie Kohler, Maxwell Krohn, David Mazieres, Robert Morris, Steve VanDeBogart, and Alexander Yip

Toward Secure Services From Untrusted Developers

Micah Brodsky[†] Petros Efstathopoulos[⋆] Frans Kaashoek[†] Eddie Kohler[⋆]
Maxwell Krohn[†] David Mazières[‡] Robert Morris[†] Steve VanDeBogart[⋆] Alexander Yip[†]
[⋆]UCLA [†]MIT [‡]Stanford

Abstract

We present a secure service prototype built from untrusted, contributed code. The service manages private data for a variety of different users, and user programs frequently require access to other users' private data. However, aside from covert timing channels, no part of the service can corrupt private data or leak it between users or outside the system without permission from the data's owners. Instead, owners may choose to reveal their data in a controlled manner. This application model is demonstrated by Muenster, a job search website that protects both the integrity and secrecy of each user's data. In spite of running untrusted code, Muenster and other services can prevent overt leaks because the untrusted modules are constrained by the operating system to follow pre-specified security policies, which are nevertheless flexible enough for programmers to do useful work. We build Muenster atop Asbestos, a recently described operating system based on a form of decentralized information flow control [5].

1 Introduction

Since Sun, and later Netscape, developed ways for a web browser to safely run untrusted code from arbitrary Internet sites, the average user's web experience has been entirely transformed. The key step was defining and enforcing security policies that prevented most malicious behavior on the part of Java applets or JavaScript scripts. Existing flaws in these policies and their implementations have had limited consequences in the wild, and now JavaScript is an essential part of the web experience. What if *server* applications, like client applications, could safely run untrusted code? Server-resident code has advantages in terms of bandwidth, latency, and simplicity. A service that allowed arbitrary users to extend it could take advantage of open-source programmers around the world, leading to new development models and applications.

Unfortunately, the consequences of any breach in a server application are dire, leading potentially to massive data loss, corruption, denial of service, or the embarrassing release of users' private information. As a result, existing attempts at server extensibility expose only a fraction of the server's resources and private data. Livejournal, for example, allows journal authors to upload sandboxed PHP renderers for their journals, but each renderer can read only a limited set of user data accessible through a strict API and has very limited write access [2].

This is one instance of a more general problem: conventional programming languages and operating system environments only offer limited tools for defining and enforcing application security policies. Desirable policies might, for example, limit an application's privilege to the minimum required to accomplish its expected task, according to the principle of least privilege [22]; or it might track the progress of sensitive information through an application, preventing its undesired escape [4, 14]. In order to constrain a program to follow such a policy, today's application programmer must cobble together combinations of existing features, such as Unix's *chroot* facility and Perl's variable tainting. The result is complex, hard to maintain, and difficult to truly secure [11].

Recent work shows that decentralized information flow control (DIFC), however, can enforce many security policies cleanly and reliably, either in the programming language [16] or in the operating system [5, 27]. These systems label all secret data and track it as it passes between software components to prevent information leaks. To enforce a security policy, an application designer decides how to assign labels to the application's data and splits the application into pieces according to the policy. For example, there might be one label per type of system user, one label per user, or one label per user per application.

Our previous work on DIFC in Asbestos improved the security of an existing application, a dynamic web site. In this paper, we move towards a new type of application previously thought too inherently insecure to build. The *wikicode* model uses flexible secure information flow control to allow loosely affiliated, and thus mutually suspicious, programmers to collaborate in constructing a secure web service. A core set of programmers design the service architecture and the corresponding security policy, as expressed in labels. Thereafter, untrusted programmers can upload binary code to the server to augment its base functionality. Since this add-on code is constrained by the system's security policies, the service still maintains its security guarantees, and in particular, untrusted code cannot inappropriately leak sensitive data from the server.

(Untrusted code can misbehave in ways not constrained by labels, of course, such as by running inefficiently, annoying the user, or simply by performing no useful function. Furthermore, while current DIFC systems eliminate overt information leaks and certain covert channels, one untrusted module might potentially extract a secret from another using a timing exploit such as wallbanging. A full system would need to address these problems as well. DIFC primitives and appropriate application security policies should be used to keep particularly sensitive data, such as private keys, from reaching untrusted code (which could in turn leak the data through covert channels). A combination of future work and existing techniques could address resource leaks and other similar problems. The work in this paper prevents high-rate overt leaks and storage channels—a necessary first step, since breaches have serious consequences and are hard to prevent with conventional techniques.)

We tested the wikicode development model by building *Muenster*, a prototype job search web site inspired by Monster.com, on top of the Asbestos operating system [5]. Job search sites serve two primary types of users: job applicants and employers. Job applicants submit their resumes and profiles to the service's applicant pool and employers search through the pool, looking for good matches. Applicant and employer data is sensitive and must be protected from inappropriate exposure: some applicants will want to restrict the employers that can see their résumés, and certain employers will want to keep their job postings secret except to select applicants. Furthermore, job search users will likely desire an endless list of features, many of which the site designer may lack the resources to implement. To satisfy this demand, Muenster users can contribute wikicode programs that implement interesting features yet cannot violate users' disclosure policies. These programs require an access control model more flexible than, for example, data partitioning, since untrusted code must compute over sensitive data.

Relative to our previous work on the Asbestos operating system [5], the central contribution of this paper is the DIFC-enabled wikicode application model, which goes beyond a more-secure variant of an existing application. To our knowledge, Muenster is the first secure application that combines both interesting cross-user data sharing policies and untrusted code. Wikicode is designed to preserve strong security policies—and even to update policies—while supporting mutually untrusted developers who are continuously developing and improving a running service. This required that we develop patterns for selective information sharing that go beyond the strict isolation and declassification presented previously. Online application development requires DIFC-aware data storage policies for service-specific data. In particular, the file system must persistently store *privilege*, allowing applications to isolate data from others while keeping it accessible to the owner across application upgrades and reboots. Finally, since wikicode authors develop on a live machine they do not control, we had to develop tools that facilitate debugging without violating the information flow rules of the system. After discussing related work and an overview of Asbestos, we discuss these contributions in turn, closing with a performance evaluation and a discussion of our experience building systems that use decentralized information flow control.

2 RELATED WORK

Several earlier systems have used confinement to safely execute untrusted programs. Web browsers and active network systems like ANT [26] execute untrusted Java [8] programs by running them in a restricted virtual machine. Browsers confine the untrusted programs by restricting the disk and network access of the Java virtual machine. The ANT execution environment further restricts untrusted code by limiting the Java language as well. Virtual machines are not limited to Java programs; virtual machine [7, 9] sandboxes can provide strong isolation for any program, including native code, for example in the NetTop project [24]. The strong isolation provided by virtual machines is unsuitable for wikicode because it precludes the safe exchange of private data between mutually untrusted programs. An untrusted wikicode program must be able to read, write and process data as long as it does not export it off the server in a way that violates the information-flow rules. A virtual machine relinquishes all control over data that it exports to other VMs for processing by another user's untrusted module. Information flow control enables wikicode to support functions like applicant searches because the applicant declassifier agents can receive secret data about employers in order to make declassification decisions, but they cannot export the secrets off the server.

It may be possible to enforce wikicode security policies at the language level using a language like JiF [16] with some modifications. The JiF compiler checks JiF source code containing inline flow control annotations to verify that sensitive information does not flow to unauthorized recipients. But this would restrict wikicode contributors to a single language and require them to disclose their source code.

Our work is an extension of the Asbestos privilege separated Web server [5], which was in turn inspired by the OK Web Server (OKWS) [10]. OKWS provides a framework for secure web services on Unix through various sandboxing techniques. Implementing wikicode as a series of OKWS services does not seem feasible, since OKWS has no way to track the data of different users even within a single service. As with virtual machine sandboxing, OKWS can prevent an untrusted program from directly reading private data, but it cannot allow an untrusted program to read private data while also preventing it from leaking that data.

Wikicode uses Asbestos, but there are a number of other mandatory access control (MAC) systems. SELinux [14] and FreeBSD [25] both include MAC functionality. We chose to use Asbestos because it efficiently supports dynamic creation of many isolation domains. This is important for wikicode because users may join the system and untrusted programs may add new security compartments at any time.

Other systems that perform automatic contamination propagation [12] include IX [15] and LOMAC [6], but these systems use predefined information data flow rules. Asbestos allows applications to define their own data flow policies.

Other MAC storage systems store labels, but they do not store privilege in the file system. Of recent systems, HiStar [27] most influenced the Asbestos persistence layer; it uses a single-level store to store instances of kernel objects, including labels. Similarly, EROS [23] stores whole system "checkpoints" on disk; system operation is resumed by reloading the last checkpointed state. The persistence layer we introduce uses the filesystem to store privilege as a regular file instead of checkpointing kernel objects to the disk. Reclaiming privilege does not assume an earlier checkpointed state and multiple process can acquire, drop and re-acquire privilege as long as they have access to the appropriate files storing it—even after a "hard" reboot.

Existing database systems [13, 20] also support per-row security labels based on users. The Asbestos database benefits from closer integration with operating system labels, allowing different processes working on behalf of one user to have different security policies. The database also allows unprivileged clients to specify the label assignments on their data. These differences allow Asbestos to support more flexible security policies.

$\star, \mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{3}$	Label levels, in increasing order
$\mathbf{T}_P$, $\mathbf{C}_P$	Process P's tracking label and clearance label
$L_1 \sqsubseteq L_2$	Label comparison: true iff $\forall x, L_1(x) \leq L_2(x)$
$L_1 \sqcup L_2$	Least-upper-bound label: $(L_1 \sqcup L_2)(x) = \max(L_1(x), L_2(x))$
$L_1 \sqcap L_2$	Greatest-lower-bound label: $(L_1 \sqcap L_2)(x) = \min(L_1(x), L_2(x))$

Figure 1: Summary of basic Asbestos label operations. Note that levels **0** and **2** are not used by Muenster.

3 Asbestos Overview

This section provides an overview of the Asbestos operating system [5] used as a base for this work. Its security and access control decisions are based on *Asbestos labels*, which control and track interprocess information flow, as well as application privilege, for an effectively unlimited number of information categories called *tags*.[1] For example, an application may choose to mark sensitive data with a tag t, which prevents any unprivileged process that has examined this data from exporting it over the network.

For each process P, the kernel maintains a *tracking label* $\mathbf{T}_P$ and a *clearance label* $\mathbf{C}_P$. The tracking label lists all of the tags P has observed, either directly or indirectly, as well as the tags for which it has privilege. When P receives a message from another process Q, its tracking label collects additional tags from Q's tracking label via a least-upper-bound operation [3] $\mathbf{T}_P \leftarrow \mathbf{T}_P \sqcup \mathbf{T}_Q$. P's clearance label governs P's ability to receive messages and protects it from unexpected contamination: the kernel silently drops Q's message unless $\mathbf{T}_Q \sqsubseteq \mathbf{C}_P$. The kernel maintains the invariant that $\mathbf{T}_P \sqsubseteq \mathbf{C}_P$.

Tracking labels and clearance labels are functions mapping tags to *levels*. This work effectively uses three levels, $\star$, **1**, and **3**. The $\star$ level represents privilege. In a tracking label, **3** indicates that a process has observed sensitive data (is contaminated with respect to the tag); in a clearance label, **3** represents the *ability* to observe sensitive data. The default level is **1**. We usually write labels using modified set notation. Thus, $L = \{a\mathbf{3}, b\star, \mathbf{1}\}$ indicates a function with $L(a) = \mathbf{3}$, $L(b) = \star$, and $L(x) = \mathbf{1}$ for $x \notin \{a, b\}$. In comparisons, $\star < \mathbf{1} < \mathbf{3}$.

Any process P can ask the kernel to allocate a new tag; this sets $\mathbf{T}_P(t)$ to $\star$. The $\star$ level is immune to contamination from received messages: $\mathbf{T}_P(t)$ remains $\star$ even after P receives a message from some Q with $\mathbf{T}_Q(t) > \star$. In information flow terms, this allows P to *declassify* information that is sensitive relative to t. Only P itself can renounce this privilege. Creating new tags is the sole mechanism by which Asbestos grants privilege; there is no tag hierarchy and, thus, no root privilege.

The kernel enforces information flow tracking by checking clearance labels and tracking labels on messaging operations, but Asbestos processes can change the labels in force for a particular message within the bounds of safe information flow. In particular, a sender can raise the levels for particular tags on a message (useful, for example, when a privileged process wants to send sensitive data), prove to the receiver that it holds privilege for one or more tags, grant privileges to the receiver, and grant clearance to the receiver. The last three cases are only possible when the sender has privilege for the relevant tags.

Asbestos services must respond to many differently-tagged requests over time. The *event process* (EP) abstraction lets such services avoid collecting tags. Event processes are limited, fast forks of a process; each event process has its own labels and address space. Various kernel structures are optimized for the case of small differences between event processes and their base process.

Asbestos was used to develop a privilege separated web server [5] inspired by the OK web server [10]. The Asbestos web server (AWS) labels user data as well as user network connections appropriately, ensuring that information leakage is not possible — e.g. by exploiting a bug in the CGI scripts or any other untrusted application component. The main AWS components include:

[1]Our notation and terminology differed in prior work.

- a set of *worker EPs*, each handling requests for a particular user and contaminated accordingly
- a set of *declassifier workers*, which allow a user to make part of her private data public.
- a trusted *demux* that accepts connections and redirects incoming requests to the appropriate worker process. For new incoming connections, it also looks up the tag identifying the user and sets up connection contamination accordingly.
- a trusted *identity daemon* process (*idd*), responsible for username to user contamination translation
- a trusted *database* service that stores a single user's privacy tag on each private row of data and propagates that tag to readers. AWS stores user data in this database.
- a trusted *network daemon* (*netd*) that labels network connections so that contaminated data cannot leak to connections without adequate clearance.

The Asbestos web server uses information flow control to protect user privacy even if a worker is compromised. It accomplishes this by labeling each user *u*'s data with a per-user tag u_T **3**. When *u* logs onto the server, AWS forks a new worker event process for the user and tags it with u_T **3** as well. AWS then disallows data tagged with u_T **3** from leaving the system towards any user other than *u*. This policy prevents leaking *u*'s data even if a worker is compromised because the operating system tracks which processes and messages contain *u*'s data.

4 A WIKICODE APPLICATION

Our wikicode demonstration application is a job search web site called Muenster, a service similar to Monster.com and Hotjobs.com. Muenster's niche is *discreet* job searching and posting: job applicants and employers have personalized control over which other users can view their information. Job search has a number of important privacy requirements. Applicants should be able to keep their resume hidden from some employers; for example, an applicant may choose to hide their job search intentions from their current employer to avoid jeopardizing their current job. Employers should also be able to keep their job opportunities confidential, except for select applicants. For example, when employers are seeking to replace high-level executives, secrecy can be critical for maintaining public relations. Employers might also want to keep their hiring techniques private. For example, on Muenster, an employer can upload their own proprietary applicant selection algorithms and Muenster will keep their algorithms hidden from other employers.

We choose this application to test wikicode because it has fairly restrictive security policies. Not all web services will be as restrictive as Muenster, but we use Muenster as an extreme test to explore the wikicode's ability to enforce security policies despite untrusted programmers adding features and code to the server. The challenge is not only to enforce the security policies on untrusted extensions, but to do so without unduly restricting their functionality.

We have built Muenster, defined its security policies, and implemented three uploaded extensions that exercise different aspects of the security system. To show how Muenster keeps user data private despite untrusted extensions running on the server, we implemented customizable user interface widgets. We also built an applicant search function that goes beyond strict isolation and declassification, and allows users to selectively share data with each other, all while running as an untrusted extension. Finally, our reference letter extension shows that an untrusted programmer can augment a running Muenster server with a new, custom DIFC security policy without interfering with Muenster or any another extension on the server.

4.1 Wikicode Server

Muenster derives its underlying security policy from the Asbestos Web server (AWS) [5]. Its design is based on the AWS, but includes several enhancements that make it suitable for wikicode development. Figure 2 illustrates the main server components.

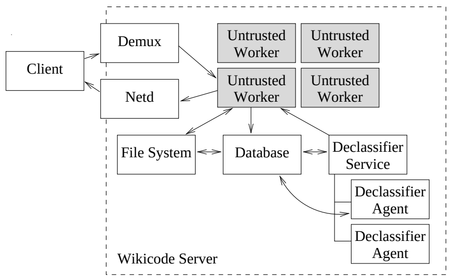


Figure 2: Wikicode server modules.

Like the AWS, Muenster assigns each user u a *data tag* called u_T and contaminates all of u's private data with $u_T\,\mathbf{3}$. The web server's network daemon prevents data leaks between users by preventing any information contaminated with one user's data tag from traveling to another user.

Since users will also store data on the server, Muenster also assigns each user a *write tag* called u_W. Only processes which run on behalf of u have privilege over u_W, and only they may modify u's data.

Untrusted Workers Usually, an untrusted extension will fit into the server as a *worker*. Each worker extension runs as a different Asbestos process; each time a different web client invokes a service, the worker forks a new event process to handle the request. The demux module grants the client's write privileges u_W to the event process and contaminates it with its data tag u_T, preventing it from leaking the client's private data. Forking a separate event process for each client prevents the worker process from accumulating tags.

The author of a worker may or may not share her worker with other users. Sensitive workers, such as a proprietary applicant matching algorithms, may be kept *private* so that no other user may invoke them. If worker author u chooses to make her worker private, she uploads the executable extension and contaminates it with her data tag $u_T\,\mathbf{3}$. Since the executable extension is tagged, the Muenster server tags any instance of the worker process with $u_T\,\mathbf{3}$, and a user that is not authorized to receive data with that tag will not be able to use the worker. If u opts to make her worker public, she uploads her executable through a privileged Muenster service which leaves the tag off the worker executable's label, allowing other users to invoke u's worker.

Allowing users to upload their own untrusted workers makes it possible for them to extend Muenster in ways the original developers may not have foreseen. We have implemented the following three example extensions as untrusted worker processes on the server to illustrate how an untrusted programmer might contribute to Muenster.

4.2 User Interface Widgets

The most basic type of untrusted extension that Muenster should support is an extension that only accesses one user's data at a time. As a concrete implementation of this policy, we developed *user interface widgets*. UI widgets give users additional options when displaying their profile to themselves when they log onto Muenster's web site. Since the widgets never need to share data with other users, they can simply run as untrusted workers.

4.3 Searching for Applicants

Untrusted user interface widgets do not need to share data between users, so it is easy to support them with strict partitioning, but in wikicode we want to allow users to share data, assuming they approve of the sharing. The challenge is to allow *selective* sharing, where users determine who can see their private data. Making the problem more challenging, a user might not know anything about the other users in the service because they might not want to reveal themselves either. Our goal was to enable users to selectively disclose their data to other users on a case by case basis, without revealing anything about the other users. To

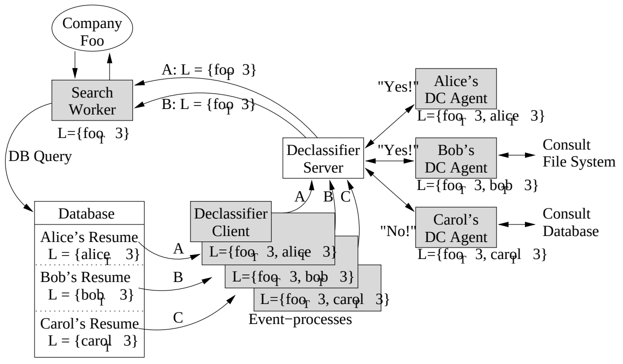


Figure 3: Declassification details for applicant search. Company Foo queries all rows, but only Alice and Bob's agents approve declassification to Foo. Clear modules are system components; shaded modules can be customized by users.

better understand this kind of controlled data sharing, we implemented a resume search in Muenster where applicants and employers can share secrets with each other, but they retain control over who learns them. This demonstrates the flexible privacy policies that an untrusted extension can implement with no special administrative server support,

In a resume search, an employer searches for job applicants and Muenster returns a set of applicant resumes. However, an applicant may want to hide her resume and the fact that she is looking for a job from some employers but not others. To help protect user privacy, a Muenster job applicant should only show up in the search results if she wants to reveal herself to the employer; otherwise, her very existence in the system should remain hidden. Like the applicants, an employer may also want to remain hidden in the system. For example, they may want to hide the fact that they are looking for employees or they may only want to show their open positions to suitable applicants.

Revealing an applicant's (tagged) resume to an employer in a mandatory access control system like Asbestos requires explicit declassification, but Muenster cannot just ask an applicant if they want to declassify their resume for employer *e*, as that would reveal that *e* is searching for applicants. Worse yet, if a notification method was used, a malicious extension could use the notifications as a covert channel to leak all the pending job descriptions, or any other private data, of a given employer.

Our approach is to use a declassification service that handles declassification requests without leaking secrets. When a client asks the declassification service to declassify some data, the trusted part of the service examines the data's label and invokes a separate declassification agent for each tag at which that label has level **3**.

Each user selects or creates their own declassification agent that may declassify the given block of data based on the user's policy. The agent may be quite powerful since it runs on the server as a regular Asbestos process. The user may select an agent that implements something simple like a blacklist or something more sophisticated, like a document relevance calculation [21] that decides based on the employer's job description. A user may even write his own declassifier agent if he chooses to. It can be written in any language, including machine code, and it can even query the database and read from the disk because the kernel will ensure that it does not violate the information flow control rules.

Figure 3 illustrates the details of an example search where employer Foo is looking for applicants. In the example, Foo logs into the untrusted *search worker* and enters some search parameters. The search worker, which is tagged with $foo_T\,\mathbf{3}$, queries the database and requests that the results be sent through declassification for recipient Foo. The database sends each resulting row, which is tagged with its owner's data tag, to a declassifier (DC) client. The DC client runs with event processes, which prevent the search worker from accumulating the tags assigned to the many database rows. With each row in a different event process, row tags do not accumulate in the DC client and the DC client can continue to process rows even if a given row is not declassified. Each DC client event process then sends its row to the DC server and asks it to remove all tags except for the untrusted worker's data tag, foo_T. Starting a new event process for each row is relatively efficient because event processes use copy-on-write support from the Asbestos kernel [5].

The DC server is trusted infrastructure, privileged with respect to all user data tags. The DC clients are not trusted, carry no privilege, and could be customized by users. For each data tag on the row, the DC server asks the corresponding DC agent if it is willing to declassify its data to user Foo. This request carries tag $foo_T\,\mathbf{3}$, preventing the DC agents from exporting information about it; this prevents the applicant from learning that the search ever occurred. For each DC agent that agrees to declassify the row, the DC server will remove the respective tag and send the row back to the original untrusted worker with the sole tag $foo_T\,\mathbf{3}$. In this example, Alice and Bob both agree to release their resumes to Foo, but not Carol.

The key is that Alice and Bob can selectively share their private resumes with Foo without learning that Foo is searching for applicants; Carol retains her privacy, but she also does not learn of Foo's search. Foo's actions are kept private unless it decides to contact Alice or Bob directly.

4.4 Reference Letters

To experiment with more sophisticated privacy policies and to see if untrusted programmers can add their own security policies to Muenster, we have implemented a reference letter feature in Muenster as an untrusted extension. The reference letter feature allows a job applicant to ask another user, such as a previous employer, to write him a letter of reference. In general, an applicant is not allowed to read his own reference letters for confidentiality reasons, and Muenster's reference letter extension enforces this. However, the extension lets an applicant configure the system to analyze the letters and withhold some letters from potential employers, without revealing any information about the letters to the applicant. The extension also ensures that a letter is only associated with an applicant if the applicant requested it. These two properties enable an applicant to retain editorial control over his reference letters without being able to read them in person or learning that a letter was ever withheld. The original Muenster developers may not have imagined such a feature, but an untrusted programmer could easily implement it, including its privacy policy, without special support from the Muenster administrators.

In the reference letter extension, an applicant issues a request to another user for a letter by inserting a row into the reference letter database table. The row contains an empty reference letter and is contaminated with both the applicant's and recommender's data tags. The applicant must configure his declassifier such that the recommender may see the row. The recommender can then write the letter and update the database row with the contents of the letter, which is still contaminated with both the applicant and the recommender's data tags. The presence of the applicant's tag means the applicant must explicitly declassify the letter before anyone can see it. When an employer searches for the letter, the applicant's declassifier can analyze the letter, possibly using sentiment detection techniques [17, 19], and remove the applicant's data tag if the declassifier approves of the letter. The letter is also tagged with the recommender's data tag, which prevents leaking the contents of the letter to the applicant. To reveal the letter to the employers, the recommender must configure his declassifier agent to remove his tag only when an employer queries the letter.

(Of course, the reference letter extension's security policy takes effect only as far as it can be enforced by labels—that is, within Muenster itself. A recommender could always post a reference letter on the web, although social pressure might discourage this. Furthermore, a recommender might always encode a message in an outwardly unremarkable letter; an applicant's declassifier could only catch obvious problems, such as overtly negative letters or letters too short to be useful. Considered generally, however, the extension (1) prevents applicants from viewing recommendations and (2) prevents recommenders from viewing applicants' potential employers, while simultaneously (3) letting applicants exclude references based on content, a difficult combination of features involving three interacting information flow tags. Similar combinations will be useful in other contexts.)

Wikicode also enables the author of the reference letter system to prevent unauthorized reference letters from entering his system by write-protecting the database table containing the letters. To do so, author a creates the reference letter table with a clearance containing his write tag, $a_W \star$, which means only processes which speak for a may modify the table. He then configures Muenster to launch his reference letter extension with his write privilege, $a_W \star$. Since a's write privilege is only given to processes that a trusts, an unauthorized extension may not add unrequested letters to the system.

The reference letter worker shows that untrusted workers have the flexibility to implement fairly sophisticated features. Because the extensions run as full Asbestos processes, and not just sandboxed processes, they have full access to the server's resources, within the information flow constraints: they can store persistent data in the file system and database without risking leaks and they can even implement their own security policies, like write protection using Asbestos labels, without interfering with the server's existing security policy.

4.5 Programming Environment

For ease of development, Muenster components may be written in its native C or in Python, a popular language for web applications. C has the advantage of high performance and Python has the advantage of development ease.

For rapid prototyping, Muenster provides a web based user interface for developing Python workers. Users edit the source code directly on the Muenster web site, click a submit button and their code is immediately available as a running Muenster service.

Muenster also provides a number of libraries and stock components in C and in Python. Declassifier libraries and worker libraries implement most of the common operations, so much of the system complexity is hidden. Untrusted programmers do not need to deal with the labeling schemes unless they need to implement their own security policy but even if they do, the extensions can be terse; the reference letter application is only 218 lines of Python, including comments.

Since the labeling scheme is consistent throughout the operating system, an untrusted extension is allowed to create and modify database tables as well as read and write file system files without the risk of leaking data.

4.6 Caveats

In its current implementation, Muenster only uses one data tag and one write tag per user. Only having one data tag per user has the limitation that all of one user's private data is in the same protection domain. Therefore, if a user is willing to declassify his resume to company Foo, then he has effectively declassified the rest of his data to company Foo. Similarly, only having one write tag per user means that that granting an untrusted extension the ability to modify a user's data allows it to modify all of that user's data. In practice, Muenster would use multiple data tags and multiple write tags per user so that users may exercise finer grained control over their private data. They would only reveal portions of their profile to other users and only grant write privileges over a portion of their data to untrusted extensions. This is possible because Asbestos allows applications to create new data tags at any time and store those new tags persistently.

5 Persistent Storage

Muenster services store addresses, resumes, and other information that must persist even if the server reboots and all volatile storage is cleared. There are several challenges in building a persistent storage system that maintains privilege and information flow invariants, even across reboots. If a contaminated process writes contaminated data to the hard disk and then later, another process reads the file, the reading process must become contaminated in the same way as the writer. Existing labeled file systems solve this problem, but the wikicode development model imposes further challenges on persistent storage. Service code may create private data and generate new security policies that apply to it. A common policy would prevent other services from modifying the data: the service generates a private t_S tag and applies a clearance of $t_S \star$ to the

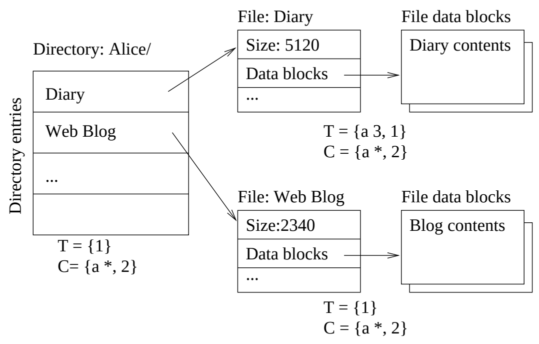


Figure 4: File and directory labels. User Alice owns a publicly readable directory *Alice*, a publicly readable file *Web Blog*, and a private file *Diary*. Only processes with privilege tag $a\star$ may modify her files.

data, whether it is stored directly in the file system or with another, database-like application. The service and the data store must therefore maintain t_S privilege when the server reboots, when the service code crashes and restarts, and even when the service code itself is updated: that is, the privilege must be made persistent, and the data store must maintain any relationships between the service's privilege and data's labels and clearances. The mechanism for preserving these relationships should be flexible enough to support arbitrary application policies. Our prior storage layer could not support this usage.

We have developed a flexible technique for preserving privileges called *pickling*, and two persistent storage services, a file system and a database. The labeled file system enables the system to store user data such as uploaded programs without the risk of leaking them to unauthorized recipients. More importantly, it also gives untrusted programs the ability to safely read and write the file system without risking privacy leaks. The persistence services uphold information flow invariants; preserve privilege across reboots; map the tag values used in one boot to those used in the next; avoid covert channels through file metadata such as names and labels; and allow applications to set up their own hierarchy of privilege. Although these requirements might be easy to provide if the file server could arbitrarily create privilege and allocate specific tags after a reboot, for higher assurance, the Asbestos file server operates within the same rules as any ordinary Asbestos process; it does not have the *option* of violating the information flow invariants.

Our file system semantics resemble those of HiStar [27], except for the way privilege is stored. HiStar and other systems such as EROS [23] avoid the problems of persistent privilege by introducing a single-level store: rebooting returns the system to a checkpointed state, and a process's tags and capabilities are stored along with its virtual memory. In HiStar's single-level store, privilege is tied to process lifetime: after the last process with privilege for a tag t dies, there is no way to recover that privilege. Our persistent store seems more familiar to most programmers and simplifies the process of recovering from application crashes without losing associated tag state.

5.1 File System Semantics

The label rules for file operations in Asbestos are similar to the label rules for processes. Each file f in the Asbestos file system has a *tracking label* $\mathbf{T}_f$ and a *clearance label* $\mathbf{C}_f$. These are analogous to the corresponding Asbestos process labels. Like a process's label, a file's label represents the contamination of the file's data. The file system contaminates any process that reads from f with its tracking label $\mathbf{T}_f$. Similarly, a file's clearance label is like a process's clearance label; a process P with tracking label $\mathbf{T}_P$ may only write to a file f if $\mathbf{T}_P \sqsubseteq \mathbf{C}_f$. For example, in Figure 4, user Alice owns the tag a and creates a file *diary* with clearance label $\mathbf{C}_{diary} = \{a\star, \mathbf{1}\}$, then the only processes that may modify *diary* are processes to which Alice grants the privilege $a\star$.

Directories have tracking labels and clearance labels exactly like regular files. Creating, renaming and removing a file are treated as writes to the directory. For example, if a process P is to create a file in directory

Operation	Label Checks and Action
read(f)	$\mathbf{T}_f \sqsubseteq \mathbf{C}_P$ Action: $\mathbf{T}_P \leftarrow \mathbf{T}_P \sqcup \mathbf{T}_f$
write(f)	$\mathbf{T}_P \sqsubseteq \mathbf{C}_f$
create(f, *dir*, $\mathbf{T}_f$, $\mathbf{C}_f$)	$\mathbf{T}_P \sqsubseteq \mathbf{C}_{dir}$, $\mathbf{T}_P \sqsubseteq \mathbf{T}_f$, $\mathbf{C}_f \sqsubseteq \mathbf{T}_f$
pickle(f, *dir*, $\mathbf{T}_f$, $\mathbf{C}_f$, f_t, f_{level}, $f_{password}$)	$\mathbf{T}_P \sqsubseteq \mathbf{C}_{dir}$, $\mathbf{T}_P \sqsubseteq \mathbf{T}_f$, $\mathbf{C}_f \sqsubseteq \mathbf{T}_f$, $\mathbf{T}_P(f_t) = \star$, $\mathbf{T}_{FS}(f_t) = \star$
unpickle(f, **3**, *password*)	$\mathbf{T}_f \sqsubseteq \mathbf{C}_P$ **Action**: $\mathbf{T}_P \leftarrow \mathbf{T}_P \sqcup \mathbf{T}_f$
unpickle(f, *level*, *password*) where $level < \mathbf{3}$	$\mathbf{T}_P \sqsubseteq \mathbf{C}_f$, $\mathbf{T}_f \sqsubseteq \mathbf{C}_P$, $level \geq f_{level}$, $password = f_{password}$ **Action**: $\mathbf{T}_P \leftarrow (\mathbf{T}_P \sqcup \mathbf{T}_f) \sqcap \{f_t\, level, \mathbf{3}\}$

Figure 5: Rules for operation on file f by process P

d, it must have a tracking label $\mathbf{T}_P$ such that $\mathbf{T}_P \sqsubseteq \mathbf{C}_d$. Also, after a process P reads the directory listing, its tracking label $\mathbf{T}_P$ will reflect any further contamination in $\mathbf{T}_d$. Figure 5 summarizes the label rules for file system operations.

Unlike process labels, file labels are immutable. Files may not be dynamically contaminated or granted privilege, and a file meant to hold a secret must be tagged appropriately when it is created. The immutable tracking label and clearance label are supplied at creation time; the file system ensures that the new file is at least as contaminated as the creating process ($\mathbf{T}_P \sqsubseteq \mathbf{T}_f$), maintaining the information-flow rules, and that the clearance label is no more tagged than the tracking label ($\mathbf{C}_f \sqsubseteq \mathbf{T}_f$). The immutable label design, which was influenced by HiStar [27], simplifies certain information flow guarantees: for example, a directory listing, which consists of the names and labels of the directory's files, has a tracking label equal to the directory's tracking label, rather than a combination of the files' labels. Designs that allow a file's tracking label to change are either more complex, leak information, or both. Although immutable labels might appear cumbersome, in practice it has not been difficult to figure out a file's intended label before the file is created.

Immutable file labels make it possible for a process to determine if it can read a file and how much more contaminated it would become by doing so. Because file labels are immutable and set at file creation time (a write to the directory), they can safely be returned when reading a directory (a read contaminates the reader with the directory tracking label). Directory read operations must not return information that might be affected by processes more contaminated than the directory itself. Thus, reading a directory reports file names, file labels, inode numbers, and the like, but not file sizes or timestamps. (A directory with $\mathbf{C}_d = \{\mathbf{1}\}$ might contain a more contaminated file with $\mathbf{T}_f = \{a\,\mathbf{3}, \mathbf{1}\}$. A process with $\mathbf{T}_P = \mathbf{T}_f$ could not write to the directory, and thus could not remove the file, but *could* write to the file, possibly changing its size.) Therefore, Asbestos provides a separate `read-size` operation that returns a file's size and also marks the reading process with the file's tracking label.

5.2 Preserving Privilege with Pickles

The main contribution of the Asbestos file system is its method to make privileges persist across system reboots. Because tag names are non-persistent and randomly generated, any persistent store must serialize tags in some form. The Asbestos file system uses *pickle files* to serialize tags.

A *pickle file*, or *pickle*, is a serialized tag represented as a file in the file system. A process with privilege for a tag may preserve that privilege by creating a pickle. Later on, another process may *unpickle* the pickle, thus acquiring the privilege that was preserved in the pickle.

To create a pickle of tag t, process P sends a request to the file system containing t and the maximum privilege (i.e., smallest level, f_{level}) that the file system should grant as the result of unpickling the pickle. Since a pickle is also a file, P also specifies its pathname, tracking label and clearance label. The file system performs the regular file creation checks on the tracking label, clearance label, and containing directory. It also confirms that P has privilege over t ($\mathbf{T}_P(t) = \star$), and that the file system process has privilege over t (the file system needs privilege with respect to every tag it pickles). Once these checks succeed, the file system can create the pickle file.

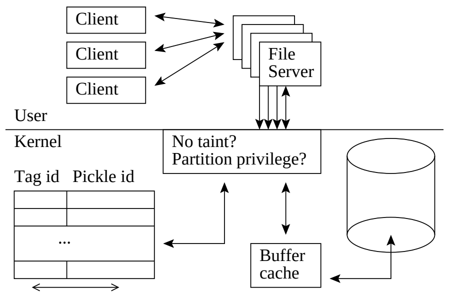


Figure 6: General file server architecture. Each client talks to a unique event process of the file server. The Event processes can access pages from the buffer cache as well as modify the tag to pickle id table.

To acquire the stored privilege in the pickle, process Q (where Q may be the process equivalent to P after one or more reboots) issues an unpickle request to the file system. The request includes the pathname of the pickle and the desired privilege level, which must be $\geq f_{level}$. The file system then checks if Q passes the normal file system checks for reading *and* writing a file with the pickle's tracking label and clearance label. The write check is done because we overload the **C** label on pickles to indicate who can unpickle them. If Q passes these checks, the file system grants t at the desired privilege level to process Q.

By starting with the simplifying assumption that there is only one persistent store, it is easy to see how pickles solve the persistence problem caused by randomly generated tags. Specifically, when trying to acquire privilege by unpickling a pickle: If the pickle was created on this boot, the file system simply returns the tag associated with the pickle. If no process has unpickled a particular pickle on a given boot, the file system simply creates a new tag for that pickle, and remembers its value for the duration of the boot.

The processes that can acquire pickled privilege can be limited by the pickle's tracking label and clearance label and the tracking labels and clearance labels on its containing directories. This means that the pickler can restrict unpickling to a set of processes that are already privileged with respect to some other tag t by putting tag t in the clearance label of the pickle. For example, if $\mathbf{T}_Q$ includes $\{t\star, p\star\}$, it could pickle p with $\mathbf{C}_Q = \{t\star, \mathbf{3}\}$. This means that a process R may only acquire p from the pickle if it has $\mathbf{T}_Q(t) = \star$. This is similar to how Alice was able to write-protect her *Diary* file in Section 5.1.

Privilege Hierarchies Pickles enable applications to construct their own privilege hierarchies using the file system and *password* protected pickles. If a key is provided during pickle creation then the correct key must also be supplied during unpickling. In order to create an independent privilege hierarchy, an application can first create a password protected pickle with an empty tracking label and clearance label to root its hierarchy. Then it would create a directory tree, protected by the first pickle, to match the privilege hierarchy it desires. After a system reboot, the application can recover the entire privilege hierarchy by unpickling the root pickle and then using it to unpickle the subsequent levels of the hierarchy; Muenster uses this technique to store its users' privilege handles.

5.3 File Server Implementation

The Asbestos file system is composed of a user level file server and two kernel components, a buffer cache and a pickle-to-tag mapping table (Figure 6). Processes access the file system by communicating with the file server, which accesses the disk through a special kernel interface.

To write data to the disk, the file server first makes sure all the contamination associated with the data has already been serialized: that is, all tags at a level other than **1** have pickle equivalents. This ensures that the file server can use the data it is about to write in a consistent way after a reboot. If there were a non-pickled tag on some file, there would be no way for the file server to figure out which tag in the fresh boot corresponded to the non-pickled tag.

The file server has read and write privileges to the raw disk blocks, so it is effectively trusted with all data in the file system and all pickled tags, but the file server is not completely privileged and is not trusted with any tags or data outside the file system. Specifically, the kernel will not allow the file server to write contaminated data to the disk. This means that the file server must have privilege for all tags on any data going to disk. As a consequence, processes may protect especially secret data from ever being written to disk by simply contaminating it with a tag that they never grant to the file server. The file system is only as privileged as it is trusted by the processes that store data in it.

A tagged process may communicate with the file server even if it has not picked all of its tags, so the file server uses event processes to avoid accumulating and spreading these tags to its clients. Each client of the file server communicates with a unique event process. Since each event process acts like a fork of the base process from the perspective of information flow, other event processes don't spread the resulting tags. However, the file server does need some information to flow between its event processes. Therefore, we extended the kernel interface to provide that information in the form of a memory-mapped buffer cache and a map between tags and pickles. Both interfaces are designed to avoid channels. For instance, writing a page to the buffer cache may communicate information between event processes, but since the kernel only allows completely uncontaminated writes, the file server cannot leak data for which it does not have privilege.

5.4 Database

We also updated our prior, memory-only database to store its data persistently and to support fully general labels. In designing the database, we used many of the design patterns found in the file system to preserve information flow constraints. For example, rows are similar to files in a directory; they each have a tracking label, $\mathbf{T}_{row}$ and a clearance label, $\mathbf{C}_{row}$. Database tables are treated similarly to directories and labels in the database are also immutable.

When processing a read query, the database returns each matching row in a separate message contaminated with the strict upper bound of the row's tracking label and the tracking label of the querying process. After all the rows are sent, the database sends a row done message with the tracking label of the process that submitted the query. The message labels ensure that all contamination tags propagate to the recipient of each row. The database sends a separate message for each row because the rows may have different tracking labels. If the rows did have different tracking labels and the database sent all rows at once, the resulting message would have a cumbersome number of contamination tags, and the recipient process would likely become too contaminated to be of further use or be unable to receive the message.

In addition to the labeling scheme, the Asbestos database differs from a conventional database because a conventional database returns one row at a time and waits for the client to request the next row. This protocol would be unusable in Asbestos because if the client is not authorized to receive one of the row messages, it would never know when to ask for the next row.

Since the database may read many rows during the course of a single query, it runs as a privileged process. For simplicity, we implemented the database by isolating a commodity database (SQLite) and adding a front-end process that handles labels. To prevent covert channels, the database only supports a subset of the SQL language; for example it does not support SQL aggregation queries like SUM.

6 DEBUGGING MECHANISMS

Muenster shows that web services can be built from untrusted components. But how can those components be built? In a conventional development scenario, a service programmer builds their service using private infrastructure, such as a development server, over which they have full control. When things go wrong with the service, the programmer can examine the entire machine, including error logs, console, and process memory. Even when physical access to the machine is not possible, developers may collect and expose debugging information (e.g. through the web browser) without any information flow restrictions.

```
1   init() {
2     tag_t t;
3     sys_new_tag(&t);
4     pickle(private_pickle_path, passwd, *, &t);
5     writefile(priv_file, C={t 3, 1}, V={t *, 3});
6     self_declassify_clearance(t, 3);
7     ...
8     sys_tag_dissociate(t, 1);
9     http_output("Initialization: success!");
10    return;
11  }
12
13  main() {
14    char * buf;
15    init();
16    ...
17    read_from_my_file(buf);
18    http_output(input);
19  }
```

Figure 7: Block of C-like code demonstrating possible bugs when developing for Muenster.

The wikicode model changes this significantly. An untrusted developer's wikicode service runs in an environment owned and built by other developers, who may not make their code public. Service developers have no privileged access, such as root or console access, since these would represent gigantic channels. Instead, service code is constrained to follow stringent security policies, which often prevent that code from exporting information, including debugging information such as backtraces and error logs. This problem is unique to information flow controlled systems.

How can a wikicode service be developed in such an environment? This section presents a set of abstractions, based on the concept of *debug domains*, which allow developers to debug their code without violating information flow rules. A debug domain represents a set of tags for which the service author has debugging privilege. Kernel errors and application behavior are exposed to the debug domain, but only as allowed by the explicit debugging privilege. A service author attempting to debug a service might, for example, create a special debugging worker, which would explicitly grant privilege to the service's debug domain. Alternately, a normal system user experiencing problems with a service might grant his or her privilege to the service's debug domain, as long as they trust the service author.

The rest of this section describes examples of errors that plague DIFC applications, the debug domain abstraction, and some ways we use debug domains to facilitate wikicode and similar development tasks.

Label Errors The observed high frequency of label errors in Muenster development, as well as their importance, made them the primary target for Asbestos debugging. Label errors may have a number of causes, including insufficient clearance to receive contamination, improper declassification and lack of privilege with respect to a faulting tag. In Muenster for instance, assume that user *Alice* develops and uploads her own module shown in Figure 7. Alice first creates a new tag, t, used to protect her application's private data, and pickles it to make t persistent (lines 3 & 4). Note that sys_new_tag() grants $t\star$ to the calling process. Then Alice creates a new file to store the private application data, sets the file labels so that readers get contaminated to $t\,\mathbf{3}$ and writers need to have $t\star$ (line 5) and raises her process's clearance label with respect to t, making it possible to receive $t\,\mathbf{3}$ contamination (line 6). Additionally, the user drops the tag from her tracking label (line 8) for two reasons: first, she doesn't need to hold "unnecessary" privilege, and second she wants to keep the process's tracking label as small as possible. Finally, the process informs the user of successful initialization by sending her a message (line 9).

This block of code will actually not work because of a label error. The first bug is on line 8: the user drops tag t prematurely. Although the user is able to read the secret file (since it has granted herself clearance to do so on line 6), she has dropped privilege to declassify information with respect to t. After reading the secret file (line 17), the process gets contaminated with $t\,\mathbf{3}$ and can no longer write data to the user's connection (line 18), since the network daemon cannot accept this tag. The call to http_output() will result in a label error due to t and there will be a warning on the console, but the user will have no idea of what went wrong.

In this case we would like the developer to receive debug information that would help resolve the error, such as the tag that caused the label error, the level of that tag in both the sender's tracking label and the receiver's clearance label, the specific type of label error, etc. Providing such debugging information directly reveals information about the sender's tracking label and the receiver's clearance label with respect to the faulting tag. This information flow should be modeled and forced to adhere to the information flow control rules of the system, to prevent any information leakage.

6.1 Debug Domains

Label errors have high impact on the development of Asbestos applications like Muenster (especially for untrusted users) since they are frequent during development and can lead to serious, complex bugs. Therefore, label errors serve as a concrete working example of our approach to debugging.

To debug a label error like the one presented in Figure 7, Alice's worker process W_A would need to explicitly notify the kernel that it wants to receive debug messages about label errors related to a set of tags, T. (In this case T consists of a single tag t.) This is done by instructing the kernel to create an internal structure called a debug domain and associate the members of T with the debug domain. Processes, such as W_A, that subscribe to the debug domain receive debug messages generated by the kernel due to errors involving one or more of these tags.[2] Adding member tags and subscribers to a debug domain affects the way information may flow and requires privilege with respect to both the debug domain and the member tags or subscriber end-points that are being added.[3]

When an error occurs because of t—or any other member of T—the kernel sends a debug message to all subscribers containing details about the offending message, including its source, destination, message ID, the tag that caused the fault, the level of the faulting handle in the sender's tracking label and receiver's clearance label, and finally the type of label error. To ensure that information flow control rules are not violated, the debug message carries the contamination of both the sender and the destination of the message that caused the label error. More specifically, a debug message resulting from a label error while Alice tries to send a message to the network daemon $Netd$ carries the label $((\mathbf{T}_{W_A} \sqcap \mathbf{T}_{Netd}) \sqcup \{t_1\star, t_2\star, \ldots t_n\star, \mathbf{3}\}) \sqcup \{p\star, \mathbf{3}\}$, where t_i are the members of T and p is the destination of the debug message, i.e. the subscriber's listening tag. In the case of Figure 7 the message would be contaminated with the label $(\mathbf{T}_{W_A} \sqcap \mathbf{T}_{Netd}) \sqcup \{t\star, p\star, \mathbf{3}\}$, since t is the only member tag of the debug domain in question. This label contaminates the receiver of the debug message with both sender and receiver contamination while declassifying information about the set of tags that are being debugged.

Using Debug Domains in Muenster Debug domains make it possible for Muenster users—including untrusted remote users—to debug their code on a "live" service, without risking data leaks, by modeling and managing debugging privilege in a decentralized fashion.

In order for user Alice to debug her application, a separate debugger process holding privilege to receive messages generated for the relevant debug domains is required to perform collection and exposure of debug information.

In Muenster users have access to one kind of processes: untrusted worker processes that are part of the web service and run on the users' behalf. Therefore Alice can use a separate, uncontaminated worker process for debugging purposes[4]. The debugging worker would need to hold debugging privilege with respect to

[2]Processes may create and subscribe to an arbitrary number of debug domains.

[3]Every debug domain is represented by a special tag, which its creator holds privilege for and can grant to other processes.

[4]A different user's worker, for example another developer who has agreed to help her debug her service, may be used for debugging purposes as long as privilege with respect to Alice's tag ($Alice_T$) is granted to that worker.

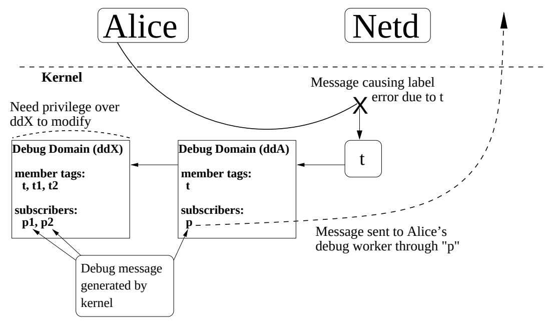

Alice
Netd
Kernel
Message causing label error due to t
Need privilege over ddX to modify
Debug Domain (ddX)
member tags: t, t1, t2
subscribers: p1, p2
Debug Domain (ddA)
member tags: t
subscribers: p
t
Message sent to Alice's debug worker through "p"
Debug message generated by kernel

Figure 8: Label error presented in the example of Figure 7. Message sent from Alice to the Network daemon (Netd) causes label error due to tag t. The kernel generates a debug message and sends it to all subscribers of the debug domains t is a member of (p from ddA and m, n from some other debug domain represented by ddX). Message sent to p is received by p's owner—in our example "Alice-debug".

Alice's relevant tags. All such debugging privilege is encoded in a debug domain: members of the debug domain are monitored for errors and debugging information is declassified with respect to members and sent to all subscriber processes—thus modeling debugging privilege as subscription to the relevant debug domain.

In practice, Alice would send a message to her debugging worker granting privilege with respect to all of her private tags that she wants to enable debugging for. Alice's debugging worker would then take the following steps:

1. Create a new debug domain ddA for Alice (and get privilege over it as the creator)
2. Subscribe to ddA (required privilege is held)
3. Add all tags granted by Alice to ddA as members (required privilege is held)
4. Optionally add $Alice_T$ to the members of ddA
5. Optionally grant Alice privilege over ddA so that she can manage it (e.g. add additional members or subscribe to it).

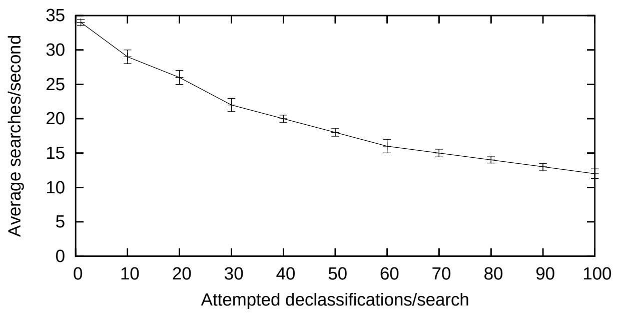
Using this setup, Alice's debugging worker will be able to receive debug messages concerning Alice. Each such message carries appropriate contamination and therefore information flow rules are not violated and data leaks with respect to non-member tags are prevented. Figure 8 illustrates an instance of these debugging mechanisms.

Debug domain generalization Debug domains are flexible enough to be applied to various debugging problems, such as system call tracing, label history tracking, and dead process tracking. System call debug messages contain the arguments to the system call as well as its return value, a label history debugger tracks changes in a monitored process's tracking label or clearance label, and a process death debugger notifies subscribers of a dying process's ID.

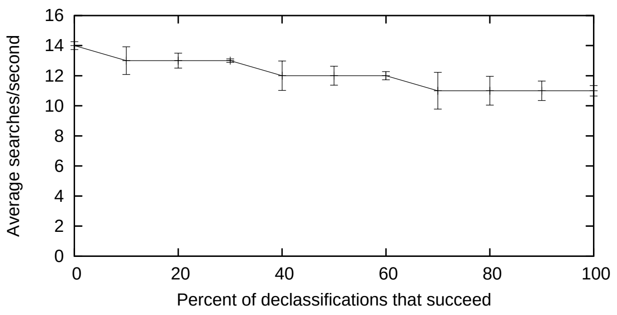
Looking back to the example code of Figure 7, if the unallocated buffer passed to the function on line 17 leads to a page fault, Alice can use a system call tracing debug domain to identify the problem. Alice may also use label history to identify the calls that led to dropping privilege (line 8) and getting contaminated (line 17) with respect to t. Finally, while debugging, one could use a debug domain to identify processes exiting early or dying due to bugs.

We have successfully used debug domains to implement debugging tools and performed debugging tests. In test cases, we have verified label error debugging in situations where the user has no console access, just like the Muenster untrusted developers.

We have also created three library calls using debug domains: $strace()$ which traces a process's system calls, $lt()$, which traces a process's tracking label changes, and the Asbestos $wait()$ library call. We have



(a)



(b)

Figure 9: (a) Completed searches per second as a function of the number of rows that the search returned (before declassification). The declassifier used for these measurements declassified fifty percent of the request. (b) Completed searches per second as a function of the percentage of declassifications that succeeded. All searches returned 100 rows from the database.

also built a simple Asbestos debugger library that allows processes to fork debuggers that collect debugging information on the processes' behalf. Furthermore, we have built a simple tool around the uploader of untrusted worker processes that would restart a Muenster worker within a debug domain, capture all debug messages the developer had clearance to receive and thus provide label error reports and system call tracing through a web interface.

7 PERFORMANCE

Muenster's declassification scheme involves a significant amount of work, so we were concerned that the performance of Muenster would be unreasonably low. To investigate the performance of the declassification system we evaluated the throughput of Muenster's search function, the heaviest user of the declassification scheme, while changing the load on the declassification system. First, we varied the query that the search worker performs to return differing numbers of results from the database. Next, we used a testing declassifier that let us change the percentage of users that declassified their data. These tests show that the performance of job searches has room for improvement, but the system is not unreasonably slow.

In these experiments, the Asbestos server was a 2.8GHz Pentium 4 with 1GB of RAM and a 7200RPM PATA drive with an average seek time of 8.5ms. The experiments took place on a gigabit local network with a Linux HTTP client generating requests. The server ran the latest version of Asbestos [1], including the persistent storage layer described in Section 5. The Muenster applicant database contained 2500 rows, each with about 30 bytes of data. Our experiments accessed data from persistent storage, but did not do any logging.

Figure 9(a) shows the number of searches that complete per unit time, as the number of rows returned by each search varies. As we expected, performance is roughly linear, with the cost of additional rows accounted for mainly by additional declassification and additional search worker processing.

Figure 9(b) indicates that the cost of declassifying (vs. rejecting the declassification request) is small. The component breakdown in Figure 10 shows mostly constant costs regardless of the fraction of rows declassified, with the exception being the search worker. The search worker formats and returns the data to the user and therefore is expected to do more work as more rows are declassified. The search worker is also written in Python which is not very efficient.

In summary, per-row declassification impacts request performance by a factor of roughly three. Though significant, this cost does not rule out the approach.

8 DISCUSSION

The Muenster application and wikicode in general reaped two key benefits from using information flow control (IFC). First, confined, automated declassification is a more expressive mechanism than simple static ACLs or capabilities. For instance, Muenster users can specify access control policies over employers they're

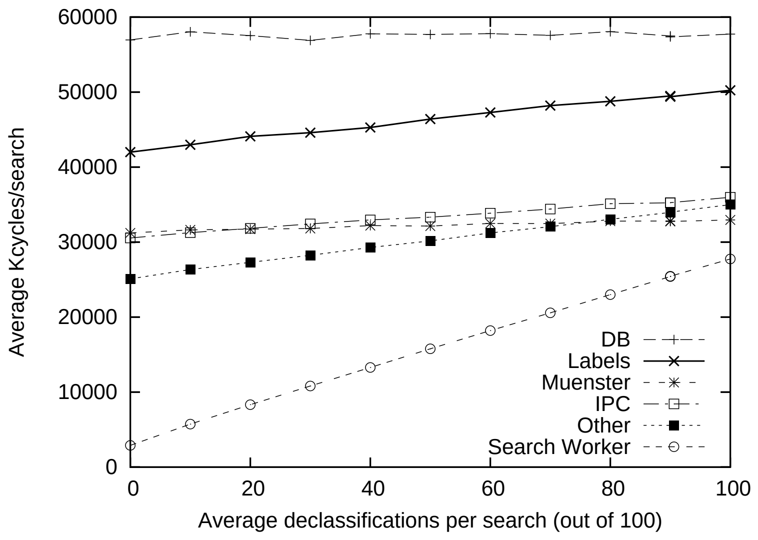


Figure 10: The cost of different components in the system, as a function of the percentage of rows that the declassifier permits. All searches returned 100 rows from the database.

not even allowed to know are in the system. More importantly, IFC provides a precise yet high-level language for specifying our end-to-end security policies. We built Muenster by envisioning desired security policies, then trying to implement them using labels. When we had difficulty in making this translation, we often discovered holes in the policy itself.

Security policies in Asbestos are specified using labels and tags, and in the system we describe, all privileged code—such as application declassifiers—make policy decisions based on labels and tags. Originally, we had planned for more complex declassifiers: perhaps a user would want to reveal their employment history more broadly than their name and address, even though both history and name were tagged identically. However, we found that these policies—which would declassify *selected* information based on the *contents* or *results* of a query, rather than declassifying *all* secret information based on the *identity* of a querier—were extremely vulnerable to covert channel exploits. A declassifier could be tricked into declassifying an applicant's name by a worker that encoded that name in a query's result using steganography. Our declassifiers thus make their decisions on the basis of information that Asbestos IFC renders unforgeable, namely labels, query history, and explicitly trusted outside information. A key lesson, also mentioned in Section 4.6, is that sources of information that need to be declassified differently must given distinct tags.

The Asbestos label mechanisms go a long way toward confining communication within the bounds of the security policy, but as in other IFC systems, covert channels are a persistent challenge. With current commodity hardware, it is possible to leak data through shared system resources [18]. Software covert channels present a more fundamental challenge, however, both because they can be replicated for high throughput and because they naturally appear in common software patterns.

Whenever information travels from one contamination level to another, covert channels can creep in. Flow control—whether there was room to enqueue a message, for example—inherently conveys information back to the source and thus cannot be used across levels. Instead, our usual solution is to place a privileged mediator process between the contamination levels. In Muenster, when data flows from lower to higher contamination, the mediator is usually the file system or the database. For example, to process a Muenster candidate search, the request query and candidate profile both climb from their previous contamination levels to the employer-candidate combined level by way of the database front end. Such privileged mediators are less confined by IFC and must be careful not to expose information leaks; for example, while developing the database front end and the declassifier service, we discovered and corrected several overlooked covert channels. In future work, it might be interesting to consider integrating finer-grained language based solutions, such as JiF [16], for these few remaining critical points in the system.

Wikicode is not immune to other covert channels, although we believe they are limited in number and mainly restricted to timing leaks. For example, declassifier agents can modulate the time they take before sending a response, yielding a covert channel to the search worker. This particular channel can be limited by bounding or quantizing the amount of time a query takes before returning, although this remains future work for us.

Another artifact of operating in the physical world is that users may try to misrepresent their identities when making accounts on Muenster, thus hampering IFC policies. To mitigate this risk, Muenster can verify user identities when they create accounts, much like financial institutions do when users create online web accounts.

9 CONCLUSION

In this paper we present *wikicode*, a new application development model that enables the co-operation of mutually untrusted developers while maintaining the application's security guarantees. Wikicode leverages the flexible information flow control model furnished by Asbestos, profiting from the power of automated declassification and the expressiveness of labels for specifying high-level security policy. Declassifiers and other confined modules can use information their creators are not authorized to see, yet they have the full power of the operating system at their disposal, without needing rigidly specified confinement policies. Modules simply pick and choose what information and services they need.

In order to support wikicode we extend Asbestos to persist privileges in a highly flexible manner, by storing them in the filesystem using the *pickle* primitive. We also add a crucial feature for introspection, the *debug domain*, allowing untrusted programmers to debug their applications without violating the security policy.

We successfully demonstrated wikicode with the Muenster application, a simple job searching site that makes heavy use of customizable, untrusted code, yet maintains the invariant that even the mere presence of employers and job seekers in the system is kept secret from other users unless explicitly disclosed. Although covert channels do remain a challenge, as yet they seem to be manageable. Finally, we showed that, in spite of the extra effort needed for all our security mechanisms, Muenster's performance is reasonable.

10 ACKNOWLEDGMENTS

This work was supported by the joint NSF Cybertrust/DARPA grant CNS-0430425 and Nokia.

REFERENCES

[1] Homepage of the Asbestos operating system, . `http://asbestos.cs.ucla.edu`.

[2] Livejournal S2 manual, . `http://www.livejournal.com/doc/s2/`.

[3] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[4] *Trusted Computer System Evaluation Criteria (Orange Book)*. Department of Defense, Dec. 1985. DoD 5200.28-STD.

[5] P. Efstathopoulos et al. Labels and event processes in the Asbestos operating system. In *Proc. 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, Oct. 2005.

[6] T. Fraser. LOMAC: Low water-mark integrity protection for COTS environments. In *Proc. 2000 IEEE Symposium on Security and Privacy*, pages 230–245, May 2000.

[7] R. P. Goldberg. Architecture of virtual machines. In *Proc. AFIPS National Computer Conference*, volume 42, pages 309–318, June 1973.

[8] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley Professional, third edition, 2005.

[9] P. A. Karger, M. E. Zurko, D. W. Bonin, A. H. Mason, and C. E. Kahn. A VMM security kernel for the VAX architecture. In *Proc. 1990 IEEE Symposium on Security and Privacy*, pages 2–19, May 1990.

[10] M. Krohn. Building secure high-performance web services with OKWS. In *Proc. 2004 USENIX Annual Technical Conference*, pages 185–198, June 2004.

[11] M. Krohn et al. Make least privilege a right (not a privilege). In *Proc. 10th Hot Topics in Operating Systems Symposium (HotOS-X)*, June 2005.

[12] C. E. Landwehr. Formal models for computer security. *ACM Computing Surveys*, 13(3):247–278, Sept. 1981.

[13] J. Levinger and R. Moran. Oracle Label Security Administrator's Guide, Mar. 2002. `http://tinyurl.com/hu4qz`.

[14] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. In *Proc. 2001 USENIX Annual Technical Conference—FREENIX Track*, pages 29–40, June 2001.

[15] M. D. McIlroy and J. A. Reeds. Multilevel security in the UNIX tradition. *Software—Practice and Experience*, 22(8): 673–694, Aug. 1992.

[16] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Computer Systems*, 9(4):410–442, Oct. 2000.

[17] T. Nasukawa and J. Yi. Sentiment analysis: capturing favorability using natural language processing. In *K-CAP '03: Proceedings of the 2nd international conference on Knowledge capture*, 2003.

[18] D. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of aes. In *Cryptology ePrint Archive, Report 2005/271*, 2005.

[19] B. Pang, L. Lee, and S. Vaithyanathan. Thumbs up?: sentiment classification using machine learning techniques. In *EMNLP '02: Proceedings of the ACL-02 conference on Empirical methods in natural language processing*. Association for Computational Linguistics, 2002.

[20] W. Rjaibi and P. Bird. A Multi-Purpose Implementation of Mandatory Access Control in Relational Database Management Systems. In *Proc. 30th Very Large Data Bases Conference (VLDB '04)*, Aug. 2004.

[21] G. Salton and C. Buckley. Term-weighting approaches in automatic text retrieval. *Inf. Process. Manage.*, 24(5):513–523, 1988.

[22] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proc. of the IEEE*, 63(9):1278–1308, Sept. 1975.

[23] J. S. Shapiro, J. Smith, and D. J. Farber. EROS: A fast capability system. In *Proc. 17th ACM Symposium on Operating Systems Principles*, pages 170–185, Dec. 1999.

[24] VMware. VMware and the National Security Agency team to build advanced secure computer systems, Jan. 2001. `http://www.vmware.com/pdf/TechTrendNotes.pdf`.

[25] R. Watson, W. Morrison, C. Vance, and B. Feldman. The TrustedBSD MAC framework: Extensible kernel access control for FreeBSD 5.0. In *Proc. 2003 USENIX Annual Technical Conference*, pages 285–296, June 2003.

[26] D. Wetherall. Active network vision and reality. In *Proc. 17th ACM Symposium on Operating Systems Principles*, Dec. 1999.

[27] N. B. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proc. 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, Nov. 2006.