Content-Based Isolation: Rethinking Isolation Policy Design on Client Systems

Alexander Moshchuk
Microsoft Research
Redmond, WA, USA
alexmos@microsoft.com

Helen J. Wang
Microsoft Research
Redmond, WA, USA
helenw@microsoft.com

Yunxin Liu
Microsoft Research Asia
Beijing, China
yunliu@microsoft.com

ABSTRACT

Modern client platforms, such as iOS, Android, Windows Phone, and Windows 8, have progressed from a per-user isolation policy, where users are isolated but a user's applications run in the same isolation container, to an application isolation policy, where different applications are isolated from one another. However, this is not enough because mutually distrusting content can interfere with one another inside a *single* application. For example, an attacker-crafted image may compromise a photo editor application and steal all images processed by the editor.

In this paper, we advocate a content-based principal model in which the OS treats content owners as its principals and isolates content of different owners from one another. Our key contribution is to generalize the content-based principal model from web browsers, namely, the same-origin policy, into an isolation policy that is suitable for all applications. The key challenge we faced is to support flexible isolation granularities while remaining compatible with the web. In this paper, we present the design, implementation, and evaluation of our prototype system that tackles this challenge.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

Keywords

Web browsers, same-origin policy, isolation

1 Introduction

Much research has been devoted to isolation *mechanisms* [38, 33, 29, 3, 53, 12, 17] to enable robust isolation containers in operating systems. However, little research has been done on the dual of the problem, isolation *policies*, namely, "what should be put into the isolation containers"; this is the topic of this paper. Isolation policy design is as critical as that of isolation mechanisms, since even with *perfect* isolation containers, an improper isolation policy can render the system *insecure*.

For example, today's Windows-based or Unix-based desktop PCs treat users as principals and protect user accounts of a machine from one another. However, mutually distrusting applications of a user can interfere with one another. This is the root cause of the significant malware problem on desktops. Learning the lessons from the desktop PCs, modern client platforms, such as iOS and Android, have refined the user-based isolation policy to treat each application code package as a separate principal, and different applications are run in separate isolation containers. This isolation policy is still too coarse because mutually distrusting content can interfere with one another within the same application. Consider the scenario where Alice opens a PowerPoint presentation from `http://alice.com/x.ppt`, which embeds a malicious video from `http://attacker.com/advideo.swf`. The attacker can compromise Alice's presentation and *all* documents being rendered by PowerPoint.

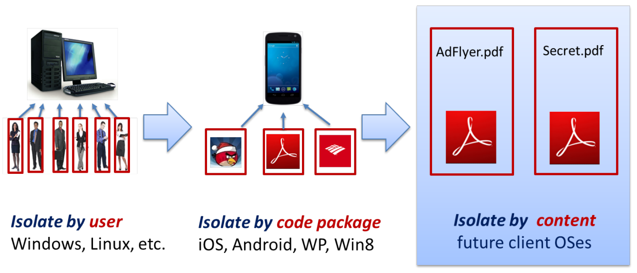

Figure 1: The evolution of isolation policies. Content-based isolation provides two benefits over other policies: (1) Any remote content's impact is minimized (2) There is no need to ask users to determine whether to risk opening remote content.

Figure 1 illustrates the evolution of isolation policies on client platforms. In this paper, we further refine the isolate-by-code-package policy and advocate future client platforms to embrace a content-based isolation policy which puts execution instances of mutually distrusting content into separate isolation containers. Content-based isolation offers a fundamentally more secure system because any content's impact including that of malicious content is *minimized* to just the content owner's isolation container. Consequently, there is *no need to ask users* to determine whether to take the risk of opening any content. Existing applications often prompt users (e.g., "Are you sure you want to open this document?") because such fine-grained containment is not available, and risk handling is outsourced to the user unreasonably.

The notion of a content-based principal model exists to some extent in today's web. All web browsers today implement the same-origin policy [41], which prevents web content of different origins (represented by a triple of <scheme, domain name, port>) from interfering with one another. Unfortunately, these principals may undesirably share the same isolation container in commercial browsers [49]. Recent new research browsers like Gazelle [49] proposed to separate web site principals into separate isolation containers, fully achieving a content-based principal model for the browser setting.

The applicability of the content-based principal model goes beyond the browser setting as modern client platforms (e.g., iOS, Android, Windows Phone, Windows 8) and applications embrace cloud-centric computing where documents and computing logic reside in the cloud and are cached on client devices. Today, applications often need to process and embed remote content. For example, Microsoft Office 2010 can open remote web documents, PowerPoint 2010 allows users to embed YouTube videos in presentations and the Google Cloud Connect [16] plug-in allows Office to sync documents with the cloud. Such applications are effectively *becoming browsers* for their own media types, and they are now facing many of the same security challenges that web browsers have faced over the past decade, namely, isolating mutually distrusting content from one another.

Today, content isolation is left as a responsibility of each application. For example, Microsoft Office 2010's new *Protected View* [30] feature provides a sandboxed *read-only* mode for documents originating from the Internet and users have to explicitly "enable editing" to remove the sandbox and its restrictions at their own risk. Adobe Reader recently introduced a similar sandboxing feature [1]. Letting each application handle content protection has serious drawbacks. First, the security of a user's cloud data is duplicated and entrusted to *all* of the user's applications. Attackers need only find one badly-written application and target it to exploit all content (at the web scale) that this application renders. Second, security logic in applications is often mixed with error-prone content processing logic; content isolation vulnerabilities are discovered not only in browsers and plug-ins [18, 49, 2], but also in desktop applications. For example, the recent RSA SecurID token compromise [36] that affected 20,000 RSA's enterprise customers was caused by a maliciously crafted Excel email attachment, and from 2008-2011, 88% (224) of Microsoft Office vulnerabilities are content parsing flaws exploitable by maliciously crafted documents [34]. Worse, many desktop applications do *not* offer any isolation for certain remote content. For example, PowerPoint 2010 renders embedded remote videos in the same process and makes no attempt to isolate them, letting potential Flash vulnerabilities endanger the PowerPoint application and its documents. Overall, users must endure weak and inconsistent security of applications that process their cloud-backed data.

In this work, we let the OS take over the burden of content isolation from applications. By consolidating content isolation logic in the OS, we *reduce* the trusted computing base from trusting many applications' isolation logic to trusting just that of the OS. The main contribution of this paper is a general content-based principal model suitable for *all* applications beyond just browsers. Our design goals are: (1) flexible isolation, from the granularity of a single addressable document to documents hosted at multiple domains, (2) compatibility with browsers' isolation policy so that attackers cannot violate browser security from non-browser applications and vice versa, and (3) easy adaptation of traditional applications.

We present a design that achieves these goals and describe our prototype system called ServiceOS, implemented as a reference monitor between the kernel and applications in Windows. We demonstrate that ServiceOS is practical by successfully adapting several large applications, such as Microsoft Word, Outlook, and Internet Explorer, onto ServiceOS with a relatively small amount of effort. Our evaluation shows that ServiceOS eliminates a large percentage of existing security vulnerabilities by design and has acceptable overhead. We also demonstrate how ServiceOS contains two working exploits.

In the rest of this paper, we describe our threat model in Section 2 and our system model in Section 3. We define our system's principals in Section 4 and show how to enforce principal definitions in Section 5. We present ServiceOS's implementation and how we adapted several traditional applications in Section 6, and evaluate ServiceOS in Section 7. We discuss related work in Section 8 and conclude in Section 9.

2 Threat Model

The primary attacker against which our system defends is the *content owner attacker*. Like the web attacker [23], the content owner attacker controls their content server(s) serving malicious data that exploits vulnerabilities or malicious code. Users may be enticed to access such malicious content from e-mail spam, malvertising, or phishing. The goal of ServiceOS is to minimize the impact of any malicious content by designing the right isolation policy and enforcement mechanisms. Our trusted computing base is the ServiceOS kernel.

We leave it up to content owners to consider network attackers, who may compromise content integrity and confidentiality. A content owner who is worried about network attackers should employ end-to-end secure channels (such as SSL) for content transport.

We are also not concerned about attackers that target specific content owners, such as cross-site scripting or cross-site request forgery attacks. These are fundamentally content-specific vulnerabilities which only content owners can fundamentally fix.

3 System Model

A principal is the unit of isolation. Program execution instances with different principal labels are isolated in separate isolation containers. We refer to an execution instance with a principal label as *principal instance* (PI).

In ServiceOS, each principal has its own local store. A user may use a certain online storage service, such as Dropbox or Google Drive. We assume that all user-downloaded content is stored on such services. This is quickly becoming the norm with modern OSes, with integrations of ChromeOS and Google Drive, Windows 8 and SkyDrive, and OS X and iCloud.

There is *no* sharing across principals or isolation containers (i.e., no global file systems unlike today's desktop systems) except through explicit cross-principal communication system APIs, analogous to IPC.

We have adopted user-driven access control [40] to allow users to share data across isolation boundaries. This is done through authentic user actions on trusted UIs (e.g., clicks on a copy button, "save-as" button, or a file picker UI) or gestures like drag-and-drop or Ctrl-V. User-driven access control enables a capability-based, least-privileged access, driven by users' natural interactions with applications and the system. We will not discuss it further in this paper and refer interested readers to [40].

For example, Microsoft Word would have its own local store on ServiceOS. A user may launch Word and start editing a new document. The document is auto-saved into Word's own local store. When the user wants to save the document to her Dropbox store (across the isolation boundary), the user clicks on a trusted "save as" button embedded in Word. The click brings up a trusted file picker window. The user then selects Dropbox, specifies the file name, and clicks on the "save" button (also part of the trusted file picker). The system only allows Word to write to a user-specified Dropbox path, but not other parts of Dropbox or other online stores, achieving least-privilege access.

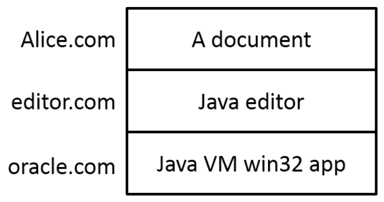

Figure 2: Execution instance as a content processing stack.

ServiceOS allows a user to have a local store, such as photo or music libraries. We label all content in the user local store as "local" principal, separate from all other principals. User-driven access control is the means for the user to get content in or out of the local store.

4 Defining Principals

An isolation policy design needs to answer two questions: (1) *how execution instances should be labeled*, or what defines a principal, (2) *how remote content is fetched and dispatched into each principal to comply with the principal definition*. This section presents our design for the former and the next section addresses the latter.

4.1 Execution instance as content processing stack

Before presenting our design on labeling execution instances, we first illustrate what constitutes an execution instance.

An execution instance may involve content from different owners. Figure 2 illustrates such an example: a document is rendered by a Java editor application which runs on a Java Virtual Machine (JVM) which in turn is a Win32 program running on Windows. The document, editor, and JVM may belong to different owners: e.g., the document may belong to `alice.com`, editor to `editor.com`, and JVM to `sun.com`. Therefore, we characterize an execution instance as a *content-processing stack*. Each layer of the stack consists of content that is owned by some entity and that needs to be addressable (for example, a web document is addressable with a URL, but user input data is not addressable). The content at a layer is consumed and processed by the next lower layer. We refer to layers below the top layer as *content processors*. For example, plug-ins in today's browsers are treated as content processors in our system.

We do *not* treat static data content as safer than active code content or content processors, because we want to allow both type-safe and native applications on our system (as is the case for most real-world client systems). Since maliciously-crafted static data can be turned into code by exploiting memory errors in native applications, we treat code and data as equally-capable content.

The content-to-processor mapping (e.g., mapping alice.com's document content to the editor) can be configured by the content owner (alice.com) or by the user. Today's web servers indicate content's MIME type using the `Content-Type` header in HTTP responses. Traditional applications can use the same mechanism to convey their content types. Additionally, we propose a new `Content-Processor` HTTP header to allow content servers to specify desired content processors by a URL or unique ID. For example, a web server serving `photo.jpg` could send:

Content-Processor: "http://www.photocompany.com/editor.app"

The content owner can sign this mapping with its private key. Users or the OS vendor can configure default content processors.

4.2 Principal labeling and isolation policies

The labels of content processing stacks ultimately determine the isolation policy of the system. Content processing stacks with the same label belong to the same principal and isolation container. We establish three goals for the principal labeling design.

1. Enable isolation policies of *arbitrary granularities* for URL-addressable resources (a file at a URL rather than an internal object in a file). The most fine-grained principal can be a single document. However, a fixed policy like this can be unnecessary and can impede functionality: some documents may not be mutually distrusting and may have intimate cross-document interactions. For example, a Microsoft Word document may interact with an Excel document intimately (e.g., by referencing its data cells or charts). So, we need a flexible mechanism to group documents into a single principal.
2. *Separate content owning from content hosting* so that a content owner can get its content hosted anywhere and be treated as belonging to the same principal. Suppose Alice created a number of documents or photos and uploaded them to various online storage services (e.g., Dropbox) or photo sharing services (e.g., Google's Picasaweb). We would want to associate all this content with the same owner Alice independent of where they are hosted. This goal is especially important for traditional applications where users often create content locally and then decide separately on where to host it. This is unlike web applications where content is usually tightly associated with its host.
3. *Be compatible with web browsers' isolation policy*. Today's web browsers' isolation policy is the same-origin policy (SOP) [41] which treats web sites as mutually distrusting principals, labeled with web site origins, a triple of $<$**protocol**, **domain**, **port**$>$ [48]. Web sites can create subdomains to have more fine-grained principals (e.g., user1.socialnet.com, user2.socialnet.com). SOP itself does not meet the above two goals that we set for principal labeling. For example, `youtube.com` and `google.com` belong to Google, but cannot be configured to belong to the same principal. SOP also does not support finer-grained isolation at path or URL level: `https://www.facebook.com/user1` and `https://www.facebook.com/user2` cannot be configured to be different principals.
   Our goal of being compatible with SOP is due to two reasons. First, it is undecidable for an OS to determine whether an application is a web browser. Even non-browser applications may use core browser components; for example, Microsoft .NET provides a web browser control which allows any .NET application to use browser functionality; similarly, iOS and Android also allow browser component inclusions in their applications. Second, even if an OS could tell the difference, it is still desirable for browsers and non-browser applications to have the same principal model because an attacker could cause browsers to render non-browser content and vice versa. Then, applications with coarser-grained isolation can be used to undermine finer-grained isolation in other applications. For example, modern OSes like iOS and Android isolate by application package (see Section 1), which is more coarse-grained than browsers' same-origin policy. Then, an application of such OSes can access two web sites of different origins and have them co-exist in the same isolation container, not meeting the web sites' expectation of being isolated from one another per same-origin policy. Therefore, we aim to design a web-compatible principal model.

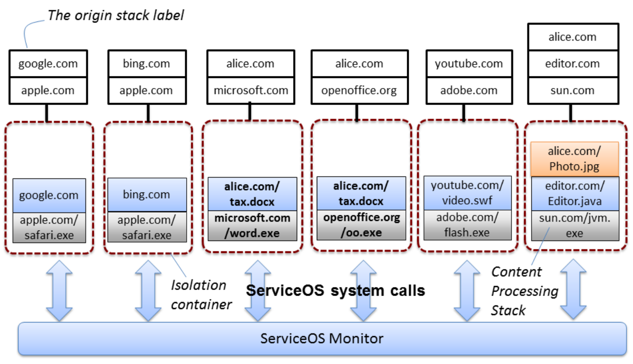

Figure 3: The origin stack as the default principal label.

4.2.1 Public key as owner ID

To achieve these goals, we propose public-key-based owner ID. Each URL-addressable content is tagged with its owner's public key and a signature that signs the host URL. This signature indicates that the signed host URL points to a resource owned by the owner of the public key. We introduce a new HTTP response header for this purpose:

Owner: publicKey=<key>; hostURLSig=signed(responseURL)

Content owners need to trust hosts to specify this header correctly and this can be easily checked by the owners. Note that ensuring content integrity and confidentiality are orthogonal features that owners and hosts can collaborate to enable in addition to principal definition specification.

A content processing stack is then labeled with a stack of public key labels, which we refer to as the *owner stack*. ServiceOS treats execution instances with the same owner stack as the same principal.

Since legacy web sites do not use our `Owner` header for principal definition, our system, by default, labels an execution instance with the origin stack of its content processing stack, capturing the origin of the content at each layer. This default gives the same isolation semantics as today's browsers: different browser vendors are content processors of different origins and isolate an origin's resources (e.g., cookie, cache, local storage) from other origins in ServiceOS as well as in today's systems.

If an `Owner` header is present for public key-based principal definition, then our system overrides the origin label with the public key value. Figure 3 shows six execution instances (or content processing stacks) and their corresponding isolation containers with the appropriate principal labels in the form of the origin stack[1]. Note that traditional browser plugins like Adobe Flash and browser renderers are treated as content processors.

Note that our owner ID design is different from locked same-origin policy [25] which uses *content host*'s X.509 certificates as principal labels for HTTPS origins. That scheme still ties the principal definition with the host. This design also differs from YURL [8] where YURL puts the host's public key as part of the URL, which also ties hosting with owning.

4.2.2 Augmenting SOP with Trust Lists

Although an owner ID offers both arbitrary isolation granularity and independence from hosts, some application developers will find it cumbersome to maintain a key pair and to compute signatures for each URL, and will resort to using the same-origin policy.

[1]We omitted protocol schemes in the origins to save space, but they should be part of the origin label.

For these developers, we introduce a "trust list" mechanism to augment SOP and to allow arbitrary isolation granularities (achieving Goal 1), but without the independence from hosts (not achieving Goal 2).

A content server can associate a trust list with any URL resource R at the server. The trust list contains a set of URLs with which R trusts to coexist in the same isolation container. This is *one-way* trust, meaning that R trusting to coexist with S does *not* mean that S trusts to coexist with R. *Two resources from two different URLs can live in the same isolation container if and only if they have mutual trust.* A resource R is allowed to be admitted to an isolation container if and only if all existing resources in the container trust to coexist with R, and R trusts to coexist with each of the existing resources.

For resources sent over HTTP, we propose a new HTTP response header called `Trust` to allow specifying a trust list. The value can be either a URL of the trust list or the trust list itself. We allow the wildcard "*" at the end of a URL for enumerating all resources at a path. We disallow wildcards for domains so that developers will not accidentally cluster mutually distrusting domains into a single principal. When the `Trust` header is missing, ServiceOS resorts back to the default, using the content server origin to label the returned resource. For resources sent over non-HTTP protocols, we resort to the default SOP where we use the application as the scheme and the IP address as the domain. We expect this to be a rare case as existing trends indicate that nearly all communication happens over HTTP [37].

The default SOP principal model is equivalent to all resources from an origin indicating a trust list of just its origin followed by a "*".

The trust list mechanism can be used to realize coarser-grained or finer-grained[2] principals than that of SOP. For example, if `youtube.com` and `google.com` want to belong to the same principal, then `google.com`'s server needs to provide the header "`Trust:list=http://youtube.com/`" *and* `youtube.com` needs to provide "`Trust:list=http://google.com/`".

Consider an example of using the `Trust` header to achieve a fine-grained principal definition. A resource, say at `http://blog.com/alice/index.html`, specifies: `Trust:list=http://blog.com/alice/*`, expressing that the resource at the URL trusts to share the container with all other content from the path corresponding to Alice. If other resources at the path also indicate the same header, then this achieves path-based principal isolation. Note that only explicitly specified URLs are trusted. In this example, the resource from `http://blog.com/` is not trusted. Similarly, individual document URLs can also be put into the `Trust` header to achieve document-level granularity of isolation.

ServiceOS enforces the `Trust` header as follows. At any time, ServiceOS maintains a stack of *effective labels* for the content processing stacks (CPS) in an isolation container. The effective label of layer L is the set of URLs of resources at layer L of all content processing stacks. Given an isolation container with an effective label stack C and and an HTTP response for a resource at URL u with a trust list, ServiceOS determines whether u should be treated as the same principal as C and be admitted to the container with

[2]Jackson et al [24] warned against using more fine-grained principal definitions than origin and claimed that the isolation boundary can break down due to (1) malicious library being included or (2) data export (e.g., form submission) being manipulated to send to attacker URLs. These two problems can also happen to the origin principal model and are not specific to finer-grained principals.

Algorithm 1 : Can the resource at URL u with a trust list be admitted to an isolation container with an effective label stack C? *If* `IsSamePrincipal`*(C, u) returns true, then ServiceOS admits u into C.*

```
 1: function IsSamePrincipal (C, u) {
 2:   us1 = C.TopLayerUrls();
 3:   us2 = {u};
 4:   repeat
 5:     if (not MutuallyTrusted (us1, us2))
 6:       return false
 7:     us1 = us1.processors
 8:     us2 = us2.processors
 9:   until (us2.processors == null)
10:   UpdateContainerLabelWithNewURL (C, u)
11:   return true
12:
13: function MutuallyTrusted(URLSet1, URLSet2) {
14:   foreach u1 in URLSet1
15:     foreach u2 in URLSet2
16:       if (u1 ∉ u2.TrustList or u2 ∉ u1.TrustList)
17:         return false
18:   return true
```

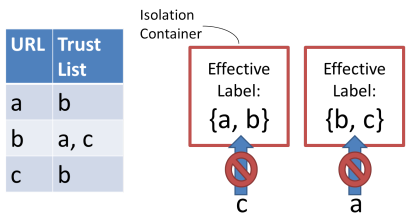

Figure 4: Non-transitivity in Trust. Although the trust lists show that a and b are mutually trusting and b and c are mutually trusting, transitive trust does not follow: a and c do not trust each other (as desired by their trust list specification) and will never share an isolation container.

Algorithm 1: ServiceOS iterates over u and its lower content processor layers and checks whether they are mutually trusted with each layer of the effective label stack C of the isolation container. If all layers of u mutually trust corresponding layers in C, u can be admitted to the container C. In that case, C incorporates u's and its content processors' labels into its own.

A interesting and desirable property of trust list is its *non-transitivity*: given that a and b are mutually trusted and b and c are mutually trusted, it does not follow that a and c are mutually trusted and can share the same isolation container. Figure 4 illustrates this property.

A content server can *easily manage* trust lists by designating a URL to contain the list and having each resource in the list use a `Trust:url=<policyurl>` header. The list (principal definition) can then be evolved without changing each resource's `Trust` header value.

For simplicity of ServiceOS logic and content server tasks, we do not mix trust-list-based principal definition with that of public-key owner. A content server should pick one to use. When both headers are present, ServiceOS uses the `Owner` header and ignores `Trust` header.

Note that `Trust` and `Owner` are applicable to only isolated content [48] (e.g., standalone program, HTML program from a web site) whose principal label is the owner of the isolated content. These headers are not applicable to library content [48], such as JavaScript included via a `<script>` tag or other libraries which do not have their own principal identity, but are designed to be included by standalone programs.

We advocate both modern client platforms (e.g., iOS, Android, Windows 8) and web browsers (and web standards) to move towards such a flexible, unified principal model.

4.2.3 Co-existence with legacy browsers

If web servers use our `Trust` and `Owner` headers to configure principal definitions, their developers need to consider their behaviors on legacy browsers.

If a principal definition is coarser-grained than an origin, a site can encounter functionality loss because legacy browsers would deny legitimate cross-origin interactions permitted by the principal definition. The site would need additional cross-origin communication code to maintain compatibility.

If a principal definition is finer-grained than an origin, then the site may lose expected isolation on legacy browsers. Security-sensitive web sites may just resort to the origin principal model to implement their isolation policies and to avoid two implementations (one for legacy browsers and one for new systems) until all major browsers adopt our proposals. Nevertheless, web sites that want to use finer-grained isolation but are not yet able to do so (e.g., `https://www.facebook.com/user1` and `https://www.facebook.com/user2` may want to be treated as different principals) may happily embrace the new headers and be safer on ServiceOS-capable systems.

5 Enforcing Principal Definitions

Central to enforcing our content-based principal definitions is how remote content is fetched from the network and to which principal instance (and isolation container) the returned content should be dispatched. Note that this enforcement mechanism is needed *in addition* to having a robust isolation container. In this section, we describe this fetching and dispatching logic and how it ensures that principal definitions are obeyed.

The dispatch decision is trivial when a Principal Instance (**PI**) of a requester principal fetches a remote resource of the same principal. We simply dispatch the returned resource to the requester PI. Determining whether a remote resource belongs to the same principal as the requester is done through the `IsSamePrincipal` check in Algorithm 1 if the resource's principal definition is based on a trust list, through public key stack comparision if an owner-ID-based principal definition is used, or through origin stack comparison if neither trust lists nor owner-based definitions are provided (Section 4).

More care is needed for cross-principal content fetch, namely, when the requester fetches content from the server of a different principal. Such a request can happen for two reasons: (1) Data communication: the requester wants some data from the responder server and the returned data should be dispatched to the requester principal instance; or (2) Spawning a new principal: the requester wants to *spawn* a new instance of the responder principal, for example if a user clicked on a hyperlink to open a new document, or a document of the requester principal embeds a resource from the responder principal; in this case, the returned data should be dispatched to the responder principal instance rather than that of the requester.

Because the requester can be malicious, we must ensure that our content fetch and dispatch logic can properly protect and isolate the responder principal from an attacking requester even in the face of arbitrary system API (ab)use.

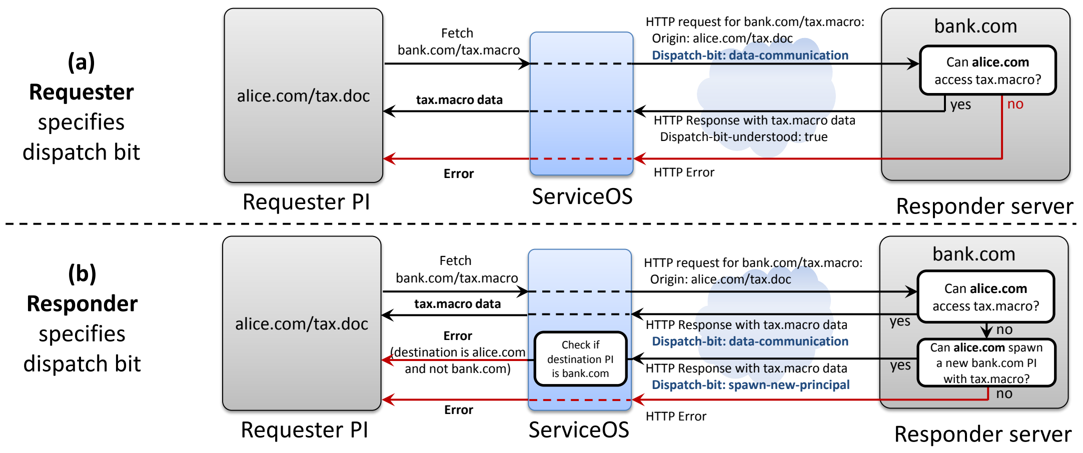
(a)
Requester specifies dispatch bit
alice.com/tax.doc
Requester PI
Fetch bank.com/tax.macro
tax.macro data
Error
ServiceOS
HTTP request for bank.com/tax.macro:
Origin: alice.com/tax.doc
Dispatch-bit: data-communication
HTTP Response with tax.macro data
Dispatch-bit-understood: true
HTTP Error
bank.com
Can alice.com access tax.macro?
yes
no
Responder server
(b)
Responder specifies dispatch bit
alice.com/tax.doc
Requester PI
Fetch bank.com/tax.macro
tax.macro data
Error (destination is alice.com and not bank.com)
Error
Check if destination PI is bank.com
ServiceOS
HTTP request for bank.com/tax.macro:
Origin: alice.com/tax.doc
HTTP Response with tax.macro data
Dispatch-bit: data-communication
HTTP Response with tax.macro data
Dispatch-bit: spawn-new-principal
HTTP Error
bank.com
Can alice.com access tax.macro?
yes
no
Can alice.com spawn a new bank.com PI with tax.macro?
yes
no
Responder server

Figure 5: Two design choices for specifying the dispatch bit. Here, a Word application rendering a document from alice.com attempts to retrieve a helper macro from bank.com, to be used in the alice.com principal instance.

The key idea in our design is to use a *dispatch bit* in either the request or the response to differentiate the two cross-principal content fetching scenarios.

There are two design choices of specifying this bit, illustrated in Figure 5. First, ServiceOS can specify the bit in the outgoing request to the responder server to convey whether the returned data will be dispatched to the requester PI or a (new) responder PI. Then, the server makes an access control decision for the request, and ServiceOS forwards the response to the requester PI. Second, the responder server (with knowledge of requester's label) can specify the bit in its response to ServiceOS to indicate whether the data is authorized to be dispatched to the requester PI or to a responder PI. Then, ServiceOS enforces the bit and either dispatches or discards the returned data. In both design choices, the request needs to contain the principal label of the requester, which we include in HTTP origin header [4].

Both designs can support legacy servers as follows. If the bit lives in the response, its absence tells ServiceOS this is a legacy server. If the bit lives in the request, then the response needs to declare that the bit was understood, the absence of that declaration tells ServiceOS this is a legacy server. For legacy servers, ServiceOS will deliver the data to the responder PI to be compatible with browsers' same-origin policy.

The advantage of putting the bit in the request is that it lets the server optimize away a response that is destined to be dropped and return an error instead. The advantage of putting the bit in the response is that it can be statically configured for each URL, rather than having the server perform access control checks for each request. Therefore, we advocate supporting the bit in both requests and responses to allow responder servers to get both advantages if desired: a diligent server can perform access control on each request and reduce network overhead (and indicate the server's knowledge of the dispatch bit in the request), and a lazy server can just statically configure its URLs when possible. When the dispatch bit from the response differs from that of request, the response's dispatch bit takes precedence.

Because HTTP has become the narrow waist of all communications [37], ServiceOS exposes system APIs for HTTP-based content fetch, and we indicate the dispatch bit with HTTP headers (Section 6). For application-level protocols other than HTTP, it is impossible (and undesirable) for ServiceOS to know their semantics and parse out their respective dispatch bits (even if they implement them). So, in the absence of dispatch bits, our system resorts to the default of returning the data to a responder PI only. With this default, applications that use peer-to-peer communication require modifications to run on ServiceOS. They must rely on explicit client-side cross-principal IPC to achieve peer-to-peer data transfer. For example, if a requester wants to retrieve data from the server of a responder principal with a non-HTTP P2P protocol, then the requester can first launch the responder's client-side PI through our system API `CreatePI(responderClientCodeURL)`; the new responder PI then retrieves the data and then passes it on to the requester PI.

Backward compatibility with existing web sites. To support web compatibility, ServiceOS supports MIME-type-based content dispatch as in Gazelle [49], translating MIME types in responses into dispatch bits in responses. We also support cookies and cross-site client-server communication primitives XMLHttpRequest Level 2 [52] and XDomainRequest [28], along with the associated CORS consent protocol [46]. Despite the availability of cookies, we strongly advocate traditional applications not to use the cookie mechanism, but to *explicitly* include their state (such as user preferences and authentication state) in their data requests to reduce the chance of cross-site request forgery (CSRF) attacks [5].

6 Implementation

We have implemented a prototype of ServiceOS on Windows 7 as a reference monitor between the OS and applications. Our implementation is in C# using .NET 3.5 and has two major components: the ServiceOS monitor and the system shell.

The monitor consists of 9.4k lines of code. It communicates with principal instances using ServiceOS system calls and upcalls, which are implemented as asynchronous XML-based messages sent over named pipes. The monitor creates a unique named pipe for each principal instance to issue system calls and receive upcalls.

The system shell consists of 3.6k lines of code. It provides a tab-based UI for users to enter URLs to visit web sites, view content rendered by traditional applications, or open standalone applications. The UI passes a newly typed URL to the monitor, which fetches the content, picks a content processor, and admits this content processing stack into the right isolation container, following the semantics of Sections 4 and 5.

Isolation mechanisms. We adopted Drawbridge [38] as our main sandboxing mechanism. Drawbridge can run unmodified Windows applications in a highly isolated mode by refactoring Windows into a library OS and virtualizing all high-level OS com-

ServiceOS API	**Examples of traditional application functions (e.g., for Word, SSH client)**	**Examples of browser functions**
CreatePI(URL, postData)	Open remote Word file	Enter URL into address bar, navigate to a link
Fetch(URL, postData)	Retrieve required resources for a document, such as templates, macros, or images	Same-origin <iframe>, <script>, <style>, original XMLHTTPRequest
Embed(URL, windowSpec, postData)	Embed remote images, spreadsheets, or videos	Cross-origin <iframe>, <object>, <embed>, <img>

Table 1: Core ServiceOS APIs

ponents, such as windowing libraries, files, or registry. Drawbridge exposes a very narrow base API for allocating virtual memory, threading, synchronization, and generic stream-based I/O (e.g., to access files that are part of an application). To protect our security policy, ServiceOS instructs Drawbridge to restrict I/O calls to only allow access to files that are part of the current principal's private storage, and to disallow pipe access other than to communicate with ServiceOS monitor.

With Drawbridge, we are theoretically able to support all user-space-only Windows applications on our system, though in practice, Drawbridge is not yet mature enough to support certain application features (such as DLLs necessary to run macros in Office documents).

Note that our system design is independent of specific isolation mechanisms like Drawbridge. As proof, we added support for another sandboxing mechanism, which associates each principal with a separate, restricted user account, and runs applications in processes using these restricted UIDs. This allows greater application compatibility than is currently possible with Drawbridge, but it is not as secure: it has a much wider API surface, and UIDs alone do not form a security boundary in Windows (e.g., one could execute shatter attacks [43] based on window messages across UID boundaries).

To complete the system, our monitor augments APIs exposed by our isolation containers with several higher-level APIs, which we discuss next.

Core ServiceOS system calls. There are three core system calls that we support: `Fetch`, `CreatePI`, and `Embed`. Table 1 shows examples of how applications may utilize these APIs. `Fetch` is used for data communications, and we implemented both synchronous and asynchronous versions. The other two calls are for creating new principal instances: `CreatePI(newUrl)` launches a new PI with a standalone UI (in a new tab), and `Embed(newUrl, <window specs>)` launches a new PI and embeds its UI window into the caller's UI. For all three calls, ServiceOS makes HTTP requests using .NET's built-in WebRequest classes. ServiceOS uses custom implementations of a cookie database and authentication manager. We implemented support for both `Owner` and `Trust` headers (Section 4), as well as algorithm 1 for admitting resources to isolation containers.

For principal definition enforcement, we have implemented the design choice of having the dispatch bit in the HTTP response since this is the case where ServiceOS has to do additional enforcement work. We add a new directive name "dispatch-to" in the existing CSP [11] HTTP header with two possible values, 'requester' or 'responder'. If a server wants the returned data to be dispatched to a responder PI, then it specifies in its response header: `X-Content-Security-Policy: dispatch-to 'responder'`. The ServiceOS monitor checks for the directive in the HTTP response for each HTTP response and performs dispatching accordingly.

Display management. Our monitor controls window positioning, dimensions, transparency, and overlaying policies but is agnostic of the UI primitives available on the host OS. It communicates these policies to the UI, which implements application windows using .NET Forms. Sandboxed Drawbridge applications run an internal RDP [31] server to expose visual output and forward user input. Our UI implements ActiveX RDP clients, which the ServiceOS monitor connects to the corresponding applications' RDP servers. When running with no isolation or with UID-based isolation, ServiceOS relies on the Windows `SetParent` API call to attach applications' UI into our shell UI and to implement `Embed()`.

Packaging content processors. Content processors and standalone applications are packaged and delivered as an archive file with extension `.app`, which our monitor decompresses and executes as a new isolated process. These packages carry a manifest file containing information like the content processor's unique ID, main executable to run, or handled content types. Web servers may allow content processors to be cached using standard HTTP headers; we rely on this as a rudimentary update mechanism and leave more elaborate, finer-grained schemes (such as [9]) as future work.

6.1 Adapting traditional applications

To run on ServiceOS, an application must connect to the ServiceOS monitor, register its display output with ServiceOS's UI, and use ServiceOS calls for fetching and dispatching content.

Some of these requirements can be handled transparently to applications. In particular, we provide a wrapper program to connect a given application to the ServiceOS monitor over a named pipe and to register its main window's visual output. As well, we observe that many Windows applications use the `WinInet` [32] library for HTTP communication. To ease porting for such applications, we used public WinInet API documentation [32]) to implement an alternate version of `wininet.dll`, which remaps its HTTP calls to invoke ServiceOS's `Fetch()` call. We then force our applications to use this DLL. We similarly remap the Windows socket library, `ws2_32.dll`, onto our raw socket API to support non-HTTP transport. Applications that do not use these libraries for communication will require porting to use our APIs, but we expect this to be rare: we examined 50 popular Windows applications, and found that all except Firefox used WinInet for HTTP communication.

This wrapping is enough to run the applications that do not fetch remote content or fetch remote content that does not need protection. As examples, we have packaged Calculator and Solitaire to run on ServiceOS. These applications can be executed simply by browsing to a URL like `http://games.com/solitaire.app` in ServiceOS's UI.

Unlike display setup and content fetch, we cannot automatically infer when to use the `CreatePI()` and `Embed()` calls to render remote content. This functionality is closely tied to application semantics and requires applications to be modified to use them. To facilitate this effort, we created a library called *LibServiceOS* which, similarly to *libc*, handles all communication details between an application and ServiceOS. It exposes ServiceOS system calls and provides an upcall interface for applications to implement. We implemented both a C++ version of LibServiceOS for native applications and a type-safe, C# version for .NET applications. In Section 7.1, we evaluate the ease of ServiceOS adaptation for several large real-world applications.

7 Evaluation

To evaluate ServiceOS, this section answers and discusses four main questions: (1) how easy is it to adapt traditional applications to run on ServiceOS, (2) by how much does minimizing impact of malicious content improve security of the system, (3) can ServiceOS stop real exploits, and (4) is our prototype's performance acceptable?

7.1 Ease of adapting traditional applications

Recall from Section 6.1 that we need to manually adapt only applications that need to spawn a new principal with `CreatePI()` (e.g., to support users clicking on a URL) or `Embed()` (e.g., to support embedding a video clip belonging to a different principal). In this section, we describe our adapation experience for several large, real-world Windows applications. Overall, we find the adaptation effort to be moderate.

7.1.1 Microsoft Word and Excel 2010

Microsoft Word is increasingly used to obtain, read, and edit remote documents. We had two goals in adapting Word: (1) isolate documents according to their content owners, and (2) allow documents to safely embed untrusted remote content, such as YouTube videos. Office 2010 exposes a rich add-in interface [39] which we used for all our modifications, thus avoiding the need to access Word and Excel source code.

Our add-in ensures that every opened document is routed to an appropriate Word instance using `CreatePI()`, and it extends Word's hyperlink class, which is used to embed links in documents, to enable an iframe-like embedding model, allowing users to embed a frame pointing to a remote web page or object. On Word's "document open" event, our add-in scans the document for these special hyperlinks, extracts information such as frame's URL, position, and dimensions, and calls `Embed()` accordingly. In response, ServiceOS will fetch corresponding remote content, dispatch it to a properly isolated principal instance, and connect the rendered content's visual output into its UI container in the Word document. As an example, we have used our add-in to securely embed video clips from YouTube, playable right from the containing Word document — functionality that has so far been unavailable in Word.

Our Word add-in consists of only 223 lines of C# code and took about one man-day to write after getting familiar with Word's add-in APIs. As a separate exercise, we have ported the plug-in to Excel 2010 to provide the same functionality. This took only 2 man-hours and resulted in a 227-line Excel add-in.

Each Word or Excel principal instance has its own UI with all menu items, most of which perform functions on the underlying document and continue to work on ServiceOS. Some functions, such as document comparison or merge, will not work if the involved documents are owned by different principals[3]. Although we have not yet done so, such features can be enabled via explicit ServiceOS-mediated IPC between different instances of Word.

7.1.2 Wordpad

Wordpad is a sophisticated text editor. We pick it as a case study of porting via source code modification, since unlike Word, Wordpad is not modularized and does not provide plug-in interfaces. It consists of more than 50k lines of C++ code, and we consider it representative of reasonably complex applications.

We extended Wordpad with the same ServiceOS support as for Word and Excel. For example, we modified the document parser to recognize special objects representing remote content and to call `Embed()`, and we modified UI code to make room for embedded content frames when rendering the document. With no prior knowledge of Wordpad, this effort took about 50 hours for one author, with most of it spent on understanding the source code. In total, we added only 435 lines of C++ code, and we expect that Wordpad developers could make these changes much more quickly. Overall, our experience showed that adaptation onto ServiceOS is feasible even when source code modification is required.

[3]Note that is an example of a future problem that never actually occurs in Office today.

	Total	**Content processing (prevented in ServiceOS)**
Microsoft Office	256	224 (88%)
Adobe Reader	202	64 (83% of known*)
Internet Explorer	144	122 (85%)

Table 2: Security analysis of vulnerabilities ('08-'11). ** We could not analyze 125 Adobe vulnerabilities with unspecified attack vectors.*

7.1.3 Internet Explorer

As our primary browser renderer, we have ported Microsoft Internet Explorer's Trident rendering engine to use our new system APIs. This effort closely mirrors the implementation of the Gazelle browser [49], so we omit further details here. In summary, we changed Trident to use ServiceOS system calls (see Table 1) using public IE COM interfaces, and by doing so, we forced Trident to use ServiceOS's isolation policies instead IE's. This could impact web compatibility. While this merits further investigation, recent work showed that an architecture like ours should have little or no compatibility hit [44].

7.1.4 Microsoft Outlook 2010

Microsoft Outlook 2010 is a popular e-mail and personal information management application. Outlook needs to isolate untrusted content in e-mail messages, but unlike Word documents or web pages, e-mail messages are not addressable via URLs. For such content, applications can still use our `Embed()` call to conveniently offload content isolation while still rendering it in-place with the rest of application's UI. We follow this approach and extend Outlook to use `Embed()` to render e-mail messages in-place as before, but in a separate protection domain, using our IE renderer (Section 7.1.3). Because ServiceOS cannot determine such content's owner information or even retrieve it, we let Outlook download message bodies and provide them directly to `Embed` (via "data:" URLs); ServiceOS uses uniquely-labeled containers in such cases.

Outlook's own protection mechanisms for restricting e-mail rendering, such as filters for <`script`>, <`iframe`>, and other dangerous tags, have been error-prone: two recent patches fixed 15 vulnerabilities that, in most severe cases, allowed attackers to take control of a system when a victim simply viewed a specially-crafted e-mail [34]. With ServiceOS, Outlook gains stronger isolation from e-mail rendering bugs, while e-mail content can benefit from additional functionality provided by IE, such as scripts and embeddable iframes.

We also extended Outlook to safely preview attachments using any ServiceOS content processor. ServiceOS picks the renderer based on the content type of the attachment; this replaces previewer lookup in the system-wide Windows registry, which is unavailable in sandboxed applications. This isolation is not only stronger but also more *usable*, as it obviates Outlook's prompting the user to consent to a preview of untrusted attachments.

Like Word, we modified Outlook using its add-in framework. This effort required 20 hours, including understanding Outlook's add-in model, and resulted in a small 342-line add-in.

7.2 Vulnerability analysis

We analyzed vulnerabilities of three applications published during 2008-2011 [34, 2], and evaluated whether ServiceOS's design mitigated them by checking whether each vulnerability was related to parsing or other content processing errors. The results are shown in Table 2. Content processing errors are widespread: 88% of Office vulnerabilities and 85% of IE vulnerabilities are related to content parsing. Adobe Reader's numbers included 125 vulnerabilities with unknown attack vectors; of the rest, 83% involved content processing. Exploits of all these flaws would be naturally contained if users were using these applications on ServiceOS; Section 7.3 demonstrates this with two concrete exploits. The rest of the vulnerabilities that ServiceOS cannot contain include insecure library loading vulnerabilities exploitable by planting malicious DLLs, HTML sanitization vulnerabilities leading to XSS, and denial-of-service vulnerabilities.

The ServiceOS monitor has only 9.4K lines of code which is significantly smaller than many applications. For example, OpenOffice has about 9M lines of code [6], and even the relatively simple Wordpad has more than 50K lines of code. Fundamentally, ServiceOS does not rely on large applications to enforce remote content security and thus reduces the TCB for isolation logic significantly.

7.3 Exploit mitigation

To verify that our system can indeed stop exploits of content processing flaws we analyzed above, we examined two real-world Word 2010 exploits. First, we used a proof-of-concept parsing exploit that uses a RTF Header stack overflow vulnerability [35] to construct a malicious document that looks for other, potentially sensitive Word documents the user has concurrently opened the same Word instance and sends them to an attacker via HTTP. The attack worked successfully on Word 2010 version 14.0.4760, bypassing both DEP and ASLR.[4]

We also crafted a second malicious document that uses macros to perform the same attack. Word treats documents opened from the web as untrusted and does not run macros by default, but offers users a choice to trust the document via a single click on a yellow security button above it. The attack document tricks the victim to click on this button by pretending to be a greeting card that needs permission to be customized. Such an attack is much easier to implement as it does not require bypasses of existing security mechanisms, and it demonstrates the pitfalls of relying on user prompts for isolation decisions. This attack works on the latest version of Word 2010, provided the victim clicks on the yellow security button.

Note that application-based isolation (such as that on iOS or Android) would also *not* be able to stop these two exploits, as they both work *within* the permission boundaries of their Word instance.

Next, we tried opening both attack documents in Word running on ServiceOS. We ran ServiceOS with UID-based sandboxing (Section 6), as Drawbridge does not yet correctly support libraries to parse an older Word document format required in the first exploit, or to run macros for the second exploit. We observed that **ServiceOS stopped both exploits**. Moreover, ServiceOS's Word version did not use any user prompts to enable macros — it no longer needs to restrict remote documents in any way since they are already isolated according to their owner. This provides better user experience for documents that legitimately use dangerous features such as macros or ActiveX.

[4]Microsoft has patched this vulnerability in later Word versions.

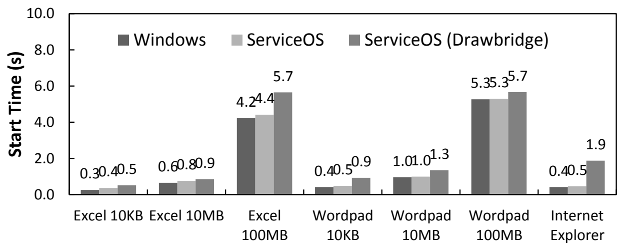

Figure 6: Time to start applications.

7.4 Performance

In measuring ServiceOS's performance, we were primarily interested in (1) startup latencies we impose on opening documents, which could happen in an existing process on Windows but require starting a new process in our model, (2) overheads on memory usage, since our model uses more processes, one for each content owner rather than one per application, and (3) performance of our content fetch APIs, which ServiceOS applications must use instead of native Windows libraries. Our measurements were performed on a 64-bit Windows 7 desktop with dual 3.16GHz Intel Xeon E8500 Duo-Core CPUs, 4GB of RAM, and a Broadcom NetXtrem Gigabit Ethernet NIC. We present results for three applications: Excel 2010, Internet Explorer (IE), and Wordpad. Excel and Wordpad experiments used 10KB, 10MB and 100MB documents; IE was used to open a simple test page on an Intranet web server. We separate the overhead of Drawbridge from overhead of the rest of our system where possible, since our system works with other sandboxing mechanisms. To run a ServiceOS application without Drawbridge, we execute it as a regular Windows process.

Startup latencies. The ServiceOS monitor and shell take 118ms to start. After a user navigates to a URL, our system starts up the appropriate renderer. Figure 6 compares this startup time to startup times of applications' native versions on Windows. We find that most overhead (up to 1.5 sec) comes from starting the Drawbridge environment. Excluding Drawbridge, in all tests ServiceOS adds less than 200ms to connect to the monitor and initialize. An obvious optimization is to maintain a small number of pre-created renderers for popular content types. Even without this optimization, we feel the startup overhead is acceptable. For example, if a user is viewing a web page with an embedded 10KB Excel spreadsheet, starting our modified Excel on ServiceOS would add only 112ms to Excel's normal startup time.

Memory usage. We measured the committed memory size for each application with one document open. Excel running on ServiceOS uses about 47MB more memory than when running on Windows, regardless of document size. This is due to Excel's loading of interoperability DLLs required to run any Excel add-in; our plug-in itself has negligible additional memory cost. Both Wordpad and ported IE carry a very small memory overhead (less than 3MB), which is required to load and initialize our 74KB LibServiceOS DLL. Drawbridge isolation introduces an additional overhead of up to 37MB for Excel.

Figures 7 and 8 show the aggregate memory usage for running multiple instances of Excel and Wordpad simultaneously. Both native and ServiceOS-enabled Excel is capable of opening multiple documents in the same process or separate processes. ServiceOS-enabled Excel render documents in the same process only if their owners are the same; e.g., a chart embedded in an Excel spreadsheet from same owner would stay in the same process. We can see that we impose no significant penalty for opening documents from the same owner, but documents from different owners (using differ-

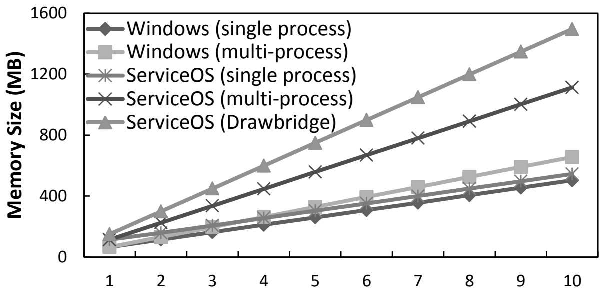

Figure 7: **Memory committed for increasing instances of Excel. Each new instance loads a 10MB document.**

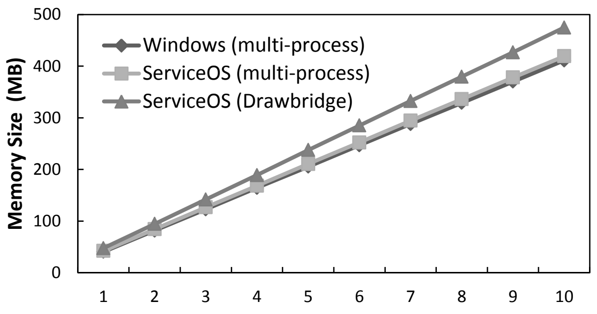

Figure 8: **Memory committed for increasing instances of Wordpad. Each new instance loads a 10MB document.**

ent processes) do carry a sizable memory overhead. However, most of it comes from (1) Drawbridge, and (2) loading Excel's add-in libraries, which adds 47MB for *each* new process. We could address (1) by picking a more efficient isolation mechanism, and improve (2) with native source-code modification, as exemplified by Wordpad which only adds 9MB for 10 instances.

ServiceOS's monitor and shell have little memory footprint, using 31.4MB of memory at worst during our tests.

Content fetch APIs. We measured the time it takes our IE renderer to use our `Fetch()` API to retrieve three documents of various sizes from a web server on a 100Mbps local network, and compared to two popular ways Windows programs retrieve content today: using .NET's native HttpWebRequest class, and using the WinInet library in Windows (recall that ServiceOS's implementation of HTTP also uses .NET's HttpWebRequest). Table 3 shows our results.

We find that ServiceOS introduces some latency for passing content to renderers, but that overhead is amortized for larger document sizes. For example, for a 10MB document, ServiceOS is only 3% (31ms) slower than a native .NET program, excluding the 62ms of Drawbridge overhead. Most of this is due to extra IPCs to ServiceOS monitor and unoptimized buffering. Interestingly, .NET's HTTP library outperforms WinInet for larger documents; the margin is large enough to mask ServiceOS's overhead. Thus, native Windows applications that used WinInet may actually encounter *faster* content fetch with ServiceOS.

We also evaluated the overhead of various new headers we introduced for content fetch in Sections 4 and 5, and found that it is negligible. For example, verifying a signature in the Owner header with a 1024-bit public key adds only about 1.5ms to each content fetch roundtrip, enforcing the *dispatch-to* directive takes less than 1ms, and parsing and verifying trust lists takes less than 1ms even when checking whether a document can be admitted to a PI with

	116 bytes	**1MB**	**10MB**
Windows (.NET program)	3ms	98ms	924ms
Native (WinInet program)	1ms	297ms	1337ms
ServiceOS, no Drawbridge	15ms	124ms	955ms
ServiceOS, with Drawbridge	16ms	156ms	1017ms

Table 3: **Overhead of fetching content of various sizes.**

100 other documents, with all documents having 101-entry trust lists.

8 Related Work

Browsers. Much recent work in browsers explored stronger isolation of web sites. OP [18] applies a microkernel architecture design with a browser kernel that enforces SOP. Tahoma [10] isolates (its own definition of) web applications using virtual machines. Major commercial browsers like Chrome and IE have adopted a process-per-tab kind of multi-process model and reduce privileges of tab processes. Gazelle [49] has a design that treats web sites as OS principals and makes its browser kernel the exclusive place for cross-principal protection. Our work builds on Gazelle and generalizes Gazelle's design to support all applications beyond web applications. We introduce the notion of a content processing stack to give a uniform treatment for both web content and content processed by traditional applications, we generalize browsers' same-origin policy to allow arbitrary isolation granularity for URI-addressable content and to enable separation of owning and hosting, and we introduce the dispatch bit mechanism to enforce principal definitions.

Modern client platforms. iOS, Android, Windows Phone, Windows 8, and the research OS Singularity [20, 51] all treat application code packages as different principals and put their execution instances into separate processes with different uids. While this marks a milestone of finally moving away from the decades-old model of treating user accounts as principals, we take another significant step by advocating a content-based principal model.

IBOS [45] aims to reduce the trusted computing base for browsers by applying microkernel design for all traditional OS components, exposing browser abstractions at the lowest software layer, and removing many components not needed by browsers. IBOS solves an orthogonal problem from ours. Our problem is to let OS provide content-based isolation for browsers and non-browser applications alike.

Embassies [19] describes web browsers as pico-datacenters where each "machine" corresponds to a web site and is isolated from other sites or "machines". This view is consistent with the semantics of existing browsers where the same-origin policy is applied to isolate web sites. Nevertheless, existing browsers do not realize isolation reliably. Embassies advocates refactoring browsers into CEI and DPI. This mirrors the refactoring done in Gazelle [49]: CEI corresponds to the Browser Kernel API in Gazelle which is runtime-independent; DPI corresponds to the runtime API in Gazelle's principal instances, which allows any programming languages or enrichment of the runtime as needed. Embassies defines web site principals using public keys, similar to one of our principal ID proposals (Section 4.2.1). Our work additionally considers content processing stacks (Section 4.1) and cross-principal content fetch of two forms: data communication and spawning new principals (Section 5) which are commonplace in practice. Finally, ServiceOS aims to support both web applications and traditional applications on the same OS platform without compromising security semantics of the web while allowing easy adaptation of traditional applications.

This work follows up on our earlier position paper on ServiceOS [50].

Other isolation policies. SubOS [22] observed that each (remote) data object needs to be rendered with a different principal label, called a sub-user id. SubOS includes a browser [21] which puts each URL page in a separate SubOS process. The SubOS's isolation policy is fixed and can be too fine-grained for many content owners. In contrast, we enable isolation policies of arbitrary granularity while being compatible with web.

PinUP [14] advocates an isolation policy that restricts which applications may access a particular local file. Unlike ServiceOS, PinUP does not isolate mutually distrusting files opened by a *single* application.

COP [7] extends the same-origin policy by letting web content specify alternate origins using unique IDs. Although the goals of COP are similar to the goals of our principal labeling proposals (Section 4.2), the underlying mechanisms are different. ServiceOS generalizes SOP to both web and native applications, while COP was designed to improve SOP's flexibility for web sites only.

Isolation mechanisms. Many mechanisms [33, 53, 12, 38, 47, 15, 29, 3, 26] have been developed to confine applications. Usually, these approaches either require applications to formulate their own security policies, resulting in many inconsistent policies coexisting on the system, or they put this burden on administrators. Our work defines a uniform isolation policy as defined by the content-based principal model and *shifts its enforcement to the OS*. We also needed to design new mechanisms for specifying (Section 4) and enforcing (Section 5) the principal definitions.

Object-capability systems and DIFC techniques [42, 54, 13, 27] can be used to implement isolation mechanisms. Realizing our principal model on these systems is an area of future research.

9 Conclusion

We advocate a *content-based principal model* in which the operating system relieves applications from the burden of isolating remotely addressable content, boosting the security of both user's data and the system by localizing the impact of any content including malicious content. Our key contribution is to generalize web browsers' same-origin policy into an isolation policy suitable for all applications while maintaining compatibility with the web. To this end, we have invented *content processing stacks* to conceptualize execution instances and introduced a general principal model that enables flexible isolation granularities using public-key-based *owner IDs* or *trust lists* to define principals. For principal definition enforcement (beyond robust isolation container design), we introduced the *dispatch bit* for cross-principal content fetch and dispatch.

We have built a substantial prototype system and adapted to it a number of real-world applications, such as Word, Excel, and Outlook. Our vulnerability study indicates that exploits against more than 80% of vulnerabilities of popular software can be contained. We have demonstrated that real-world exploits' impact can indeed be isolated in our system. Our performance evaluation shows acceptable performance overhead. From these experiences, we believe that content-based isolation policy is indispensable for future client platforms where applications increasingly interface with cloud-backed content.

10 Acknowledgements

We thank Barry Bond, Galen Hunt, and Reuben Olinsky for their help with the integration with Drawbridge. We thank Chris Hawblitzel for his valuable feedback on our paper draft. We thank Bill Barlowe for his help with our vulnerability analysis.

11 References

[1] Adobe Secure Software Engineering Team. Inside Adobe Reader Protected Mode. `http://blogs.adobe.com/asset/tag/protected-mode`.

[2] Adobe Security Bulletin Search. `http://www.adobe.com/support/security/`.

[3] AppArmor Application Security for Linux. `http://www.novell.com/linux/security/apparmor/`.

[4] A. Barth. The web origin concept. Internet-Draft, `http://tools.ietf.org/html/draft-abarth-origin-09`, 2010.

[5] A. Barth, C. Jackson, and J. C. Mitchell. Robust defenses for cross-site request forgery. In *To appear at the 15th ACM Conference on Computer and Communications Security (CCS 2008)*, 2008.

[6] Build FAQ for OpenOffice.org. `http://www.openoffice.org/FAQs/build_faq.html`.

[7] Y. Cao, V. Rastogi, Z. Li, Y. Chen, and A. Moshchuk. Redefining web browser principals with a configurable origin policy. In *DSN*, 2013.

[8] T. Close. Decentralized identification. `http://www.waterken.com/dev/YURL/`.

[9] C. Collberg, J. H. Hartman, S. Babu, and S. K. Udupa. Slinky: static linking reloaded. In *USENIX ATC*, 2005.

[10] R. S. Cox, J. G. Hansen, S. D. Gribble, and H. M. Levy. A safety-oriented platform for web applications. In *IEEE Symposium on Security and Privacy*, 2006.

[11] Content security policy (csp). `https://wiki.mozilla.org/Security/CSP/Spec`.

[12] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch. Leveraging legacy code to deploy desktop applications on the web. In *OSDI*, 2008.

[13] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Zieglar, E. Kohler, D. Mazieres, F. Kaashoek, and R. Morris. Labels and Event Processes in the Asbestos Operating System. In *SOSP*, 2005.

[14] W. Enck, P. McDaniel, and T. Jaeger. Pinup: Pinning user files to known applications. In *ACSAC*, 2008.

[15] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications. In *USENIX Security*, 1996.

[16] Google Cloud Connect for Microsoft Office. `http://tools.google.com/dlpage/cloudconnect`.

[17] GreenBorder. `www.google.com/greenborder/`.

[18] C. Grier, S. Tang, and S. T. King. Secure web browsing with the OP web browser. In *Proceedings of the IEEE Symposium on Securiy and Privacy*, 2008.

[19] J. Howell, B. Parno, and J. Douceur. Embassies: Radically refactoring the web. In *NSDI*, 2013.

[20] G. Hunt and J. Larus. Singularity: Rethinking the Software Stack. In *Operating Systems Review*, April 2007.

[21] S. Ioannidis and S. M. Bellovin. Building a secure web browser. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, 2001.

[22] S. Ioannidis, S. M. Bellovin, and J. M. Smith. Sub-operating systems: A new approach to application security. In *SIGOPS European Workshop*, 2002.

[23] C. Jackson. Improving browser security policies. PhD thesis, Stanford University, CA, 2009.

[24] C. Jackson and A. Barth. Beware of Finer-Grained Origins. In *Web 2.0 Security and Privacy*, May 2008.

[25] C. Karlof, J. Tygar, D. Wagner, and U. Shankar. Dynamic Pharming Attacks and Locked Same-Origin Policies for Web Browsers. In *CCS*, 2007.

[26] T. Kim and N. Zeldovich. Making Linux protection mechanisms egalitarian with UserFS. In *Usenix Security*, aug 2010.

[27] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information Flow Control for Standard OS Abstractions. In *21st Symposium of Operating Systems Principles*, 2007.

[28] E. Lawrence. Xdomainrequest - restrictions, limitations and workarounds. `http://blogs.msdn.com/b/ieinternals/archive/2010/05/13/xdomainrequest-restrictions-limitations-and-workarounds.aspx`.

[29] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of the 2001 USENIX Annual Technical Conference*, 2001.

[30] Microsoft. Protected View in Office 2010. `http://blogs.technet.com/b/office2010/archive/2009/08/13/protected-view-in-office-2010.aspx`.

[31] Microsoft. Remote desktop protocol. `msdn.microsoft.com/en-us/library/cc240445(PROT.10).aspx`.

[32] Microsoft. Windows Internet API. `msdn.microsoft.com/en-us/library/aa385331(VS.85).aspx`.

[33] Microsoft. Windows Vista Integrity Mechanism Technical Reference. `http://msdn.microsoft.com/en-us/library/bb625964.aspx`.

[34] Microsoft security bulletins and advisories: MS10-087, MS10-079, MS10-103. `http://www.microsoft.com/technet/security/current.aspx`.

[35] MS Office 2010 RTF Header Stack Overflow Vulnerability. `http://www.exploit-db.com/exploits/17474/`.

[36] P. Muncaster. How We Found the File That Was Used to Hack RSA, August 2011.

[37] L. Popa, A. Ghodsi, and I. Stoica. Http as the narrow waist of the future internet. In *HotNets*, Monterey, CA, 2010.

[38] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt. Rethinking the library OS from the top down. In *ASPLOS*, 2011.

[39] Programming application-level add-ins. `http://msdn.microsoft.com/en-us/library/bb157876.aspx`.

[40] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, and H. J. Wang. User-driven access control: Rethinking permission granting in modern operating systems. In *IEEE Symposium on Security and Privacy*, 2012.

[41] J. Ruderman. The Same-Origin Policy. `http://www.mozilla.org/projects/security/components/same-origin.html`.

[42] J. S. Shapiro and S. Weber. Verifying the eros confinement mechanism. In *IEEE Symposium on Security and Privacy*, 2000.

[43] Shatter Attacks - How to break Windows. `http://www.thehackademy.net/madchat/vxdevl/papers/winsys/shatter.html`.

[44] K. Singh, A. Moshchuk, H. J. Wang, and W. Lee. On the incoherencies in web browser access control policies. In *IEEE Symposium on Security and Privacy*, 2010.

[45] S. Tang, H. Mai, and S. T. King. Trust and protection in the Illinois Browser Operating System. In *OSDI*, 2010.

[46] A. van Kesteren. Cross-origin resource sharing. W3C Working Draft, `http://www.w3.org/TR/cors/`, 2010.

[47] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *SOSP*, 1993.

[48] H. J. Wang, X. Fan, J. Howell, and C. Jackson. Protection and Communication Abstractions in MashupOS. In *ACM Symposium on Operating System Principles*, October 2007.

[49] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The Multi-Principal OS Construction of the Gazelle Web Browser. In *USENIX Security*, 2010.

[50] H. J. Wang, A. Moshchuk, and A. Bush. Convergence of Desktop and Web Applications on a Multi-Service OS. In *HotSec*, 2009.

[51] T. Wobber, A. Yumerefendi, M. Abadi, A. Birrell, and D. R. Simon. Authorizing Applications in Singularity. In *Eurosys*, March 2007.

[52] XMLHttpRequest Level 2. `http://www.w3.org/TR/XMLHttpRequest/`.

[53] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *IEEE Symposium on Security and Privacy*, 2009.

[54] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *7th Symposium on Operating Systems Design and Implementation*, 2006.