Convergence of Desktop and Web Applications on a Multi-Service OS

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

Alexander Moshchuk
University of Washington, Seattle
anm@cs.washington.edu

Alan Bush
Microsoft Corporation
alanbush@microsoft.com

Abstract

A paradigm shift has been taking place in the personal computer sharing model: a computer is no longer shared by users, but shared by mutually distrusting applications or other content. This *multi-application* sharing model is mismatched with today's *multi-user* operating systems like Windows and Linux, which offer protection only across users. This mismatch contributes significantly to today's malware problem: a user is often tricked to download and install malware which runs with the privileges of the user or even with escalated privileges to harm the user's machine.

Web-centric computing is another significant trend in computing, which makes web browsers a dominant client application platform. The browser platform supports a multi-application sharing model. However, today's web browsers have never been designed and constructed as an operating system: different web site principals may coexist in the same protection domain, and there is no coherent support for resource access, control, and sharing. This makes browsers a vulnerable and functionally limited platform.

In the light of these two trends, we envision ServiceOS, a multi-service OS on which web applications and traditional desktop applications converge. "Service" comes from "Software-as-a-Service". A service is some generic content which can be either code or data. Services are hosted in the cloud and cached on the client. The owner of the service is an OS principal. ServiceOS will enable an application model that synthesizes the best elements from both desktop and web applications, providing fundamentally better security without sacrificing functionality. We sketch our design and present open challenges for this new paradigm of computing.

1 Introduction

Over the past two decades, there has been a paradigm shift in the personal computer sharing model. As illustrated in Figure 1, a computer is no longer used and shared by multiple users, but rather it is used by a *single* user to render content from different Internet origins or to run multiple, sometimes mutually distrusting applications, many of which are obtained directly from the web. The old model was commonly referred to as a *multi-user* system where users are the principals of the system. We call the new model a *multi-application* system, where the owners of the applications are the principals. Since a principal is the unit of protection, a system must ensure that each principal's resources are protected from one another in terms of both resource access and usage. For example, one principal's state should not be accessed by another principal, and one principal's use of a shared resource, such as CPU, should not interfere with (e.g., deny the service to) another principal.

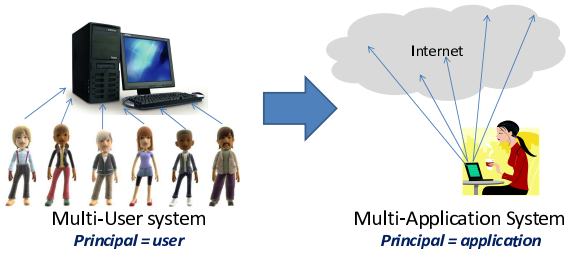

Figure 1: **Multi-User vs. Multi-Application systems**

Despite the multi-application sharing model in today's personal computing, today's operating systems like Windows and Linux do not treat applications (or content from different origins) as first-class principals for protection and resource management. Applications, even when downloaded by a user from obscure remote hosts, run with the privileges of the logged-in user without protection from one another. This contributes significantly to today's malware problem: a user is often tricked into downloading malicious software, which then typically runs with the user's privileges (and sometimes with escalated privileges), steals user's data, and interferes with other software on the computer.

In parallel to the transformation of the computer sharing model is the trend of web-centric computing. As significant amount of functionality has been created and shifted into the web; web browsers are quickly becoming a dominant client application platform. Often, users only need a browser — independent of the underlying OS, device, or their physical location — to satisfy all of their computing needs from information search to shopping, banking, communication, office tasks, and entertainment. The same-origin policy (SOP), the central security policy for web browsers, mandates that a web document (or a web application) from one origin cannot interfere with another document from a different origin; the origin is defined as the triple of <protocol, DNS domain name, port> [17]. This implies that a web applica-

tion is treated as a principal on the browser platform and that its origin labels the principal [19]. However, today's browsers have never been designed and constructed as an OS for these web applications: different principals may coexist in the same protection domain, and there is no coherent support for resource access, control, and sharing. The lack of OS design in browsers impedes web applications' capabilities of interacting with devices (e.g., camera, GPS) and allow a misbehaving (whether malicious or poorly-written) web application to monopolize system resources to itself.

As we can see from these two trends, traditional OSes no longer have the right definition of OS principals for today's computing needs, while browsers have the right principal model, but they lack an OS design. We envision that traditional desktop applications and web applications will *converge* on *ServiceOS*, a multi-*service* OS. "Service" comes from "software-as-a-service". A service is some generic content which can be either code or data. Services are hosted in the cloud and cached on the client. The owner of the service is an OS principal. In ServiceOS, web applications and traditional desktop applications are equal citizens. ServiceOS will enable an application model that synthesizes the best elements from both desktop and web applications: it gives controlled access to all system resources, enables offline operations, provides proper protection across application principals in terms of both access and usage, and supports the same software and content distribution model as web applications, which enables OS, device, and location independence.

With ServiceOS, we can achieve backward compatibility for web applications. Adapting legacy desktop applications may sacrifice backward compatibility when the applications rely intimately on cross-application sharing through file systems or registry, which is permitted on today's multi-user OSes.

The key benefits of ServiceOS are: 1) Our multi-service OS design minimizes the impact of malware, which would run as a separate principal from other applications and is therefore constrained by the malware's protection domain. Although many sandboxing mechanisms [15] exist to contain malware, a user is required to actively apply them to untrusted applications. In contrast, ServiceOS sandboxes all services by default. 2) By providing resource access, control, and sharing, ServiceOS enhances web applications' functionality and quality to be on par with desktop applications.

2 Limitations of today's browsers and operating systems

In this section, we elaborate on why today's browsers and operating systems are insufficient to support our converged desktop and web application scenario.

2.1 The lack of OS design in today's browsers

Web browsers have never been built as an operating system, which is manifested as follows.

Cross-principal protection. For a long time, browsers used only a single process for executing all web site principals. Even with the latest released browsers IE 8 [11] and Google Chrome [3, 16], which employ multiple processes, an arbitrary number of web site principals can coexist in a single process. For example, when a site `a.com` embeds other principals, such as `ad.com` and `gadget.com`, the site and the embedded principals coexist in the same process in IE 8 and Chrome. As a result, principal protection logic is intertwined with content processing code at the content object and method level. This is extremely error-prone as manifested in numerous cross-principal vulnerabilities [4, 13, 2]. The experimental OP browser [9] also doesn't provide complete cross-principal protection; for example, OP offers no protection of display among principals. Our work Gazelle [20], which is a part of the ServiceOS effort, addresses cross-principal protection at the OS level and completely segregates principals into separate protection domains.

Device access and control. Today's browsers don't expose any APIs for accessing devices like GPS or camera. This significantly impedes web applications' capabilities. On the other hand, browsers offer an extremely permissive plugin model, allowing plugin software to extend the browser and directly access the underlying OS. Many plugins then expose devices to plugin content. For example, Adobe Flash and Google Gears enable access to certain devices and provide their own respective security policies. This model is flawed because the plugins' security policies are completely disconnected from one another and from the browser's policy, making it impossible to enforce a uniform policy across all application and content types. Worse, a plugin compromise leads to a compromise of all privacy-sensitive devices and all other principals [14, 18], since today's plugins are granted unrestricted access to the underlying OS.

Resource scheduling. Today's browsers administer no control on resource sharing, such as CPU scheduling and network bandwidth allocation, among web site principals when resources are under contention. As a result, principals can interfere with one another in resource usage, leading to poor service quality and allowing misbehaving principals to monopolize system resources. For example, a malicious, embedded advertisement principal can get an unfair amount of CPU, memory, and network bandwidth, and even cause denial-of-service to the host page.

2.2 Insufficiency of existing operating systems

Web applications and traditional desktop applications have two fundamental differences: trust model and cross-principal application composition. These differences reveal limitations of existing operating systems in supporting our converged application scenario. We discuss these differences and limitations below.

Different trust model. Existing operating systems give significantly more trust to desktop applications than web applications. For example, desktop applications are allowed to access devices and file systems, but web applications are constrained by the browser sandbox, which denies such access. The reason may be that desktop application installations and launches are explicitly triggered by the user. In contrast, web applications need not be installed and can silently launch other web application principals, for example by embedding them in frames or objects, without any user interaction.

Nevertheless, even explicit user actions like installation and application launch do not necessarily mean that the application is trustworthy. Social engineering attacks have exploited users to install and launch malware, which has been a serious and prevalent security problem [8]. Therefore, in ServiceOS's converged application model, we trust neither web applications nor desktop applications by treating each application as a separate principal.

One question is whether we can map each application to a separate user principal in an existing OS and leverage an existing OS's access control mechanisms for resource or device access. Unfortunately, device access control mechanisms in existing OSes have poor manageability, as each physical device needs to be configured with an access control policy regardless of its semantics. With a large and growing number of principals involved, each addition or upgrade of a physical device will require reconfiguring access control. This is cumbersome and error-prone.

Cross-principal application composition. Another significant difference is that web applications often embed applications from different principals. This is a popular form of mashup that has enabled many creative web sites. Such an application composition practice renders CPU (or other resource) scheduling in today's OSes unsuitable, even when each application is mapped onto a separate user principal. CPU scheduling in today's OSes would treat each principal instance as a scheduling unit and fairly share the CPU among all involved instances, regardless of the principal instance embedding hierarchy. This is undesirable in that an embedded application has the ability to embed an arbitrary number of principals and thereby monopolize resource usage. Figure 2 shows such an example. An application `a.com` embeds an ad application from `attackerAd.com`. By embedding many other principals, `attackerAd.com` can deny resources to `a.com`.

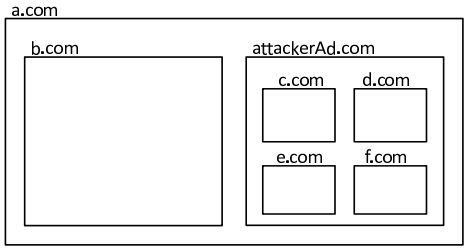

Figure 2: **A Web Service Composition Scenario.** By embedding many services, `attackerAd.com` mounts a denial-of-service attack against `a.com`.

Note that cross-principal application composition can be useful for desktop applications as well. For example, a Word document can reference and embed an Excel spreadsheet. On today's desktops, such embedding happens to documents of a single user. It is easy to imagine the desirability of such composition in a cross-principal fashion.

3 The ServiceOS framework

3.1 Application and principal model

So far, we have used the term "application" loosely and interchangeably with "content". Traditionally, an application is a software program. However, a software program can take input which can in turn be a program. For example, a Java virtual machine software takes Java programs as input; the Microsoft Word software takes Word files as input, which may contain macros. This program and input relationship can continue recursively. For example, a Java program may be a Jython [1] interpreter that takes Python programs as input and so on.

Note that the input data of a program may have a different owner from that of the program. For example, a Word document may be owned by a user who bears no relationship to Microsoft, which created and may even host the Word software.

Therefore, we need to clearly define the principal of an execution instance and the trust relationship between various owners involved. To this end, we generically use *content* to refer to either a software program or static data, and *content renderer* to refer to a software program that processes or renders some content. For example, a Java VM is a content renderer for Java content. A content renderer is itself a form of content which can be further rendered by some other content renderer. Content can indicate which content renderer it trusts to render it; this mapping can also be configured by the user or the OS vendor. For example, in a web browser, the content renderer for traditional web content, such as HTML, JavaScript and CSS content, is the browser's ren-

dering engine which parses HTML and CSS, interprets JavaScript, and maintains the DOM objects. The mapping between the content and its renderer in this case is achieved by web servers indicating the content's MIME type and by the browser's built-in configuration. We anticipate the length of the content rendering chain to be typically short with length one or two.

An execution instance of such a *rendered-by* chain is driven by the content at the head of the chain. For example, a Jython program drives the execution of the Jython interpreter, which drives the Java VM, which drives the x86 runtime. Therefore, the principal needs to be the owner of the head content. For web content, it is the SOP origin of that content. For a traditional application's content, it can be a combination of the content's SOP origin and its certificate. Figure 3 gives an illustration of our application and principal model for a traditional web site, traditional plugin content, x86 content, and Word content.

This application and principal model is inspired by today's web and browsers. In addition to the default rendering engine of a browser for rendering traditional web content, other content renderers can be added to the browser in the form of plugins to render other content. For example, the content renderer for Adobe Shockwave video content is the Adobe Flash plugin. In fact, the HTML object tag can be readily used for expressing the chain of content and content renders. Suppose a user Alice is hosting her tax form at `https://alice.com/tax.html`, which contains the following content:

```
<object data=''https://alice.com/tax.docx'' classid=''msword''>
    <param name=''template''
           value=''https://templates.com/tax.dotx''>
</object>

<object id=''msword''
        data=''https://microsoft.com/software/Word.exe''
        classid=''Xax''>
</object>
```

The first object element describes the actual tax document in the MS Word format. The content renderer is indicated in the "classid" attribute. The Word template file (for styling) is given as a parameter. The second object element describes MS Word as content and indicates the content renderer to be Xax, which is a sandboxed execution runtime for x86 code [6].

Content embedding is a powerful paradigm in today's web, which has enabled creative application compositions. Content can be embedded as *isolated* content, the principal ID of which is the origin of embedded content [19]. For example, the frame tags are used to embed isolated content . Alternatively, a content can be embedded as *library* content, the principal ID of which is the includer's origin [19]. For example, the script tag is used to embed library content. We adopt this content embedding model in ServiceOS. Content renderers need to express to ServiceOS which content types are isolated and which are library. ServiceOS is then responsible for creating new principals or routing content to the right principals.

Some research OSes, such as Singularity [21], and recent commercial platforms, such as iPhone or Android [5], treat "applications" as principals. The "application" refers to a software program, and doesn't have our notion of recursive content and content renderer chain. This leads to a fundamental difference in the principal definition. For example, `alice.com/tax.docx` and `bob.com/tax.docx` will be rendered in the protection domain of a single principal because they both use the application MS Word. This is undesirable in that these two documents actually come from mutually distrusting origins. In fact, rendering any malicious Word document can compromise MS Word and all the documents that will be rendered in the future. In contrast, in ServiceOS, these two documents will be rendered as two separate principals in different protection domains with a MS Word instance running in each protection domain.

3.2 Units of protection, failure containment, and resource allocation

A principal is the *unit of protection*. In ServiceOS, principals are completely isolated in resource access and usage by default, which is consistent with the same-origin policy in today's browsers. Any sharing across principals must be made explicit.

Just as in desktop applications, where instances of an application are run in separate processes for failure containment and independent resource allocation, a principal instance in ServiceOS is the *unit of failure containment* and the *unit of resource allocation*. For example, when a user enters the same URL in different tabs, two instances of the same principal are created. Principal instances are isolated for all runtime resources, but principal instances of the same principal share persistent state such as cookies and other local storage.

Protection unit, resource allocation unit, and failure containment unit can each use a different mechanism or the same mechanism depending on the system implementation. Nevertheless, the resource allocation unit and the failure containment unit need to be the same or more fine-grained than the protection unit. For example, if content renderers are implemented in native code, then an OS process can be the suitable mechanism for all three purposes. If content renderers are written in type-safe code, a software-based process, such as SIP [10], may be used for the unit of protection, and a more fine-grained unit may be used for resource allocation and failure con-

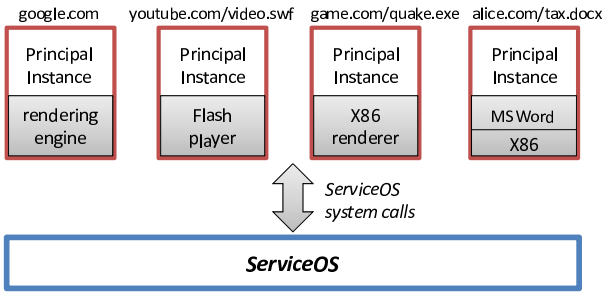

Figure 3: **The ServiceOS Architecture.** Each gray box corresponds to a content renderer needed to render the content above it. The content's SOP origin is labeled on top of each principal instance. Principal instances interact with system resources through the ServiceOS system calls.

tainment.

3.3 Architecture

Figure 3 shows the high-level architecture of ServiceOS. ServiceOS is completely agnostic to any content or application semantics and is relatively simple. Content processing is done in the unprivileged principal space. Principal instances access system resources (e.g., network, storage, display, devices, IPC) through ServiceOS system calls. The system calls are designed for web applications' needs and include content fetches, window or display delegation, and the APIs typically needed by desktop applications and desired by web applications, such as device access and control. ServiceOS is the trusted computing base and exclusively enforces system-wide security policies including the same-origin policy, cookie access policies, and remote server access policies (e.g., XMLHttpRequest). This applies even to plugin software, unlike today's browsers. These policies must be made compatible with existing browsers: existing web applications should not run less securely on ServiceOS, and new ServiceOS-enabled web applications should not run less securely on legacy browsers.

Now we give a brief security analysis on the architecture. Malicious content can compromise only its own protection domain. A malicious content renderer compromises the protection domains of all content that utilizes the renderer. A vulnerable renderer compromises only the protection domains of the content that exploits the renderer's vulnerabilities.

4 Research challenges

4.1 Resource access control

As discussed in Section 2, resource access control in browsers is largely non-existent and the current practice of exposing underlying OS to plugins results in disparate security and access control policies. Furthermore, device access control in traditional OSes poses poor manageability. We identify the following requirements and challenges for resource access control in ServiceOS:

No application is trusted: In ServiceOS, no application is trusted. Therefore, all privacy-sensitive resources like location or microphone must be denied access by default.

Uniform access control for all content types: ServiceOS needs to exclusively manage the resource access control independent of the content types or content renderers in the principal space. This is unlike today's browsers which allow different access control policies for different plugins.

Independence from physical devices: To address the manageability problem in existing OSes'access control, access control needs to be managed over resource semantics such as location rather than physical devices such as a specific model of a GPS device.

Usability: The user is the root of trust who gives applications permissions to access resources. We must ensure our access control is usable. Existing approaches, such as that of Android and Facebook, presents an application manifest at the application installation time showing all the resources needed by the application; then the user is presented with the choice of accepting or denying the application. This approach may be too coarse-grained. We are exploring a new kind of manifest that presents *must-have* and *prefer-to-have* resources and enables the user to deny prefer-to-have resources for privacy reasons. Other considerations in usable access control include understanding the tradeoffs of asking for user permissions at the installation time or right before resource usage, and managing the lifetime of a permission, whether it is forever, per-session, for per-use. Another challenge is not to overburden users with too many prompts.

4.2 Resource sharing

As pointed out in Section 2, cross-principal content embedding renders commodity OSes' resource scheduling unfair. In ServiceOS, when under resource contention, we must ensure that an embedded principal will not monopolize resources arbitrarily. In addition to fairness, we need to meet the requirements of special applications, such as that of real-time multimedia streams or mission-critical functions like telephone usage.

4.3 Cross-principal sharing

Unlike traditional OSes, ServiceOS imposes strict isolation across its principals by default. Instead of burdening programmers with application-specific data access and control, it is interesting to investigate systematic sharing and information flow control that may be offered by ServiceOS. Enabling application developers to easily spec-

ify and validate sharing will be crucial for secure and robust application engineering. Techniques from HiStar [23], Flume [12], or Asbestos [7] may be tailored for the ServiceOS setting.

4.4 Web application backward compatibility vs. security

ServiceOS allows backward compatibility with existing web applications. However, the security policies in today's browsers are not coherent and often at conflict, resulting in significant difficulty in engineering secure web applications. An important research topic could be to design a set of security policies that are coherent and analyze their compatibility cost.

4.5 Painless porting of desktop applications and plugin software

Desktop applications and plugin software would need to be ported to ServiceOS. Xax [6] and NaCl [22] have given some evidence of the feasibility of porting traditional applications. Nevertheless, these systems still require significant effort to port sophisticated desktop applications; it would be useful to investigate tools to make this process easier.

5 Summary

In this position paper, we have proposed ServiceOS, a multi-application operating system that supports both web and desktop applications as first-class principals. This platform captures the multi-application sharing nature of today's personal computing, and it embraces web-centric computing by providing web applications with long-needed OS support, including cross-principal protection and resource management. ServiceOS closes a significant source of malware that plagues today's personal computers. At the same time, ServiceOS enriches web applications to be as capable as desktop applications. Such a platform scenario also raises many open research challenges.

References

[1] The jython project. `http://www.jython.org/`.

[2] Security advisories for Firefox 2.0. `http://www.mozilla.org/security/known-vulnerabilities/firefox20.html`.

[3] A. Barth, C. Jackson, C. Reis, and T. G. C. Team. The security architecture of the Chromium browser, 2008. `http://crypto.stanford.edu/websec/chromium/`.

[4] S. Chen, D. Ross, and Y.-M. Wang. An Analysis of Browser Domain-Isolation Bugs and A Light-Weight Transparent Defense Mechanism. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2007.

[5] Developing secure mobile applications for Android. `http://www.isecpartners.com/files/iSEC_Securing_Android_Apps.pdf`, October 2008.

[6] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch. Leveraging legacy code to deploy desktop applications on the web. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, 2008.

[7] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the Asbestos operating system. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, 2005.

[8] E. Grandjean. A prime target for social engineering malware. `http://www.mcafee.com/us/local_content/misc/threat_center/msj_prime_target.pdf`, 2008.

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

[10] G. Hunt, C. Hawblitzel, O. Hodson, J. Larus, B. Steensgaard, and T. Wobber. Sealing os processes to improve dependability and safety. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, March 2007.

[11] What's New in Internet Explorer 8, 2008. `http://msdn.microsoft.com/en-us/library/cc288472.aspx`.

[12] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. *SIGOPS Oper. Syst. Rev.*, 41(6):321–334, 2007.

[13] Microsoft security bulletin. `http://www.microsoft.com/technet/security/`.

[14] Microsfot Security Intelligence Report, Volume 5, 2008. `http://www.microsoft.com/security/portal/sir.aspx`.

[15] D. S. Peterson, M. Bishop, and R. Pandey. A flexible containment mechanism for executing untrusted code. In *Proceedings of the 11th USENIX Security Symposium*, 2002.

[16] C. Reis and S. D. Gribble. Isolating web programs in modern browser architectures. In *EuroSys*, Nuremberg, Germany, March 2009.

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

[18] Symantec Global Internet Security Threat Report: Trends for July - December 07, April 2008.

[19] 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.

[20] H. J. Wang, C. Grier, A. Moshchuk, S. King, P. Choudhury, and H. Venter. The multi-principal OS construction of the Gazelle web browser. In *Proceedings of the 18th USENIX Security Symposium*, 2009.

[21] T. Wobber, A. Yumerefendi, M. Abadi, A. Birrell, and D. R. Simon. Authorizing applications in singularity. *SIGOPS Oper. Syst. Rev.*, 41(3):355–368, 2007.

[22] 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 *Proceedings of the IEEE Symposium on Security and Privacy*, 2009.

[23] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *OSDI '06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, 2006.