In-Nimbo Sandboxing

Michael Maass, William L. Scherlis, and Jonathan Aldrich
Institute for Software Research
Carnegie Mellon University
{mmaass, wls, aldrich}@cs.cmu.edu

ABSTRACT

Sandboxes impose a security policy, isolating applications and their components from the rest of a system. While many sandboxing techniques exist, state of the art sandboxes generally perform their functions within the system that is being defended. As a result, when the sandbox fails or is bypassed, the security of the surrounding system can no longer be assured. We experiment with the idea of in-nimbo sandboxing, encapsulating untrusted computations away from the system we are trying to protect. The idea is to delegate computations that may be vulnerable or malicious to virtual machine instances in a cloud computing environment.

This may not reduce the possibility of an in-situ sandbox compromise, but it could significantly reduce the consequences should that possibility be realized. To achieve this advantage, there are additional requirements, including: (1) A regulated channel between the local and cloud environments that supports interaction with the encapsulated application, (2) Performance design that acceptably minimizes latencies in excess of the in-situ baseline.

To test the feasibility of the idea, we built an in-nimbo sandbox for Adobe Reader, an application that historically has been subject to significant attacks. We undertook a prototype deployment with PDF users in a large aerospace firm. In addition to thwarting several examples of existing PDF-based malware, we found that the added increment of latency, perhaps surprisingly, does not overly impair the user experience with respect to performance or usability.

1. INTRODUCTION

Sandboxes are the most common way to secure systems and components that are currently intractable to verify and that we cannot trust. Application sandboxing is a technique used to impose a security policy on an application. Rather than assuring the compliance of an application or a computation with a policy, a sandbox is constructed to encapsulate the application or computation and any malicious behavior it may manifest. The sandbox implements a security policy on control and data flows to reduce the likelihood and consequences to a target system of malicious code execution within the sandbox. Sandboxing is also useful for increasing confidence in the execution of applications that are trusted but that are nonetheless vulnerable, due to complexity or other factors.

But what happens when the sandbox fails? This paper presents and evaluates an application-focused sandboxing technique that is intended to address both sides of the risk calculation – mitigating the consequences of traditional sandbox failures while also increasing the effort required by an attacker attempting to compromise the target system. Our technique is reminiscent of software as a service, thus allowing us to evaluate the security benefits of those and similar architectures. We present the technique, describe a prototype we developed to support a field trial deployment, and assess the technique according to a set of defined criteria. Here is a summary of our hypotheses regarding the in-nimbo technique for application sandboxing:

- Attack Surface Design Flexibility: In-nimbo sandboxing provides flexibility in attack surface design. We focus on tailoring the sandbox to the application, which doesn't allow for a "one size fits all" implementation. Our technique allows architects to more easily design and implement an attack surface they can confidently defend when compared to other techniques. This is because the technique is less constrained by structures within an existing client system.

- Attack Surface Extent: Our technique results in encapsulated components with smaller, more defensible attack surfaces compared to the cases where the component is encapsulated using other techniques. Along with the previous criterion, this should have the effect of diminishing the "likelihood" part of the risk product.

- Consequence of Attack Success: Remote encapsulation reduces the consequences of attack success. Our technique reduces the magnitude of the damage resulting from an attack on the encapsulated component when compared to the same attack on the component when it is encapsulated using other techniques. That is, we suggest that our approach diminishes the extent of consequence in the risk product.

We also apply the following criteria:

- Performance: We focus on latency and ignore resource consumption. Our technique slightly increases the user-

HotSoS '14 Raleigh, NC USA

perceived latency of an encapsulated component compared to the original version of the component. This is based on data from our field-trial deployment, and in this regard we do benefit from the fact that the encapsulated component is a large, complex, and relatively slow vendor application.

- Usability: We focus on changes to the user experience as well as ease of deployment. Our technique does not substantially alter the user's qualitative experience with the encapsulated component. Deployment is straightforward, as described below.

As alluded to above, we evaluate these points based on data from a field trial with disinterested users at a large aerospace company. Performance measurements were taken from this deployment. We should note that we were not able to control all features of the deployment, and there was some architectural mismatch with respect to communications channels and cloud architecture. Additionally, our system is an operationalized prototype with some inconvenient design choices. The data we obtained nonetheless suggests that our approach offers real benefits – even in the presence of these limitations.

In sections below, we detail the design of the sandbox architecture and the prototype we developed. The principal user-visible change is in the form of window chroming as seen by the user; this is a consequence of expedient features of our prototype design.

We also present two thought experiments to illustrate the potential flexibility our technique provides when designing the attack surface. Finally, we present a structured comparison between our implemented approach and a mainstream approach to highlight the points above regarding the risk product – controlling and minimizing the extent of the attack surface and diminishing the consequences of attack success.

Researchers have been working on encapsulation techniques since at least 1993, when Wahbe[56] introduced the term sandboxing to describe software-based fault isolation (SFI) as a means of preventing distrusted modules from escaping their fault domains. Since then a plethora of sandboxing techniques have been introduced, including mainstream uses such as the Java Runtime Environment in the 1990's and Microsoft's Protected Mode for Internet Explorer 7 in late 2006.

Modern applications that have been sandboxed are typically complex to a degree that defies most existing techniques for finding and fixing vulnerabilities. Sandboxing has helped architects make an application's attack surface smaller and more defensible. Essentially, the architect increases the attacker's costs by applying a security layer to exposed computations. It is particularly valuable for applications that have been compromised repeatedly, are an advantageous target to an attacker, or that cannot be verified using existing methods. In doing so, the defender effectively reduces the verification problem to that of verifying the sandbox.

In practice, sandboxes tend to combine a number of distinct encapsulation techniques to achieve their goal. For example:

- chroot sandboxes (commonly referred to as chroot jails) redirect the root of the filesystem for a specific application [23]. This redirection has the effect of preventing the application from accessing files that are not below what it sees as the filesystem root.

- Google NaCl applies SFI and runtime isolation to prevent memory corruption exploits in native code and to constrain what the native code can do at runtime respectively [57].

- Microsoft Internet Explorer's Protected Mode works by constraining the execution of risky components, such as the HTML render, using rules encoded as integrity levels [12].

- TxBox intercepts all of the system calls made by an application and ensures the calls do not violate a security policy [31].

- TRuE intercepts systems calls, employs SFI, and redirects system resources [31].

The combination of techniques applied by each sandbox varies both the types of policies that can be enforced by the sandbox [19, 26] and how usable the sandbox is for the architect applying it. A chroot jail cannot prevent the execution of arbitrary code via a buffer overflow, but chroot jails can make sensitive files unavailable to the arbitrary code and are quick and easy to deploy when compared to applying Google NaCl to a component. Of course, these two sandboxes are not intended to encapsulate the same types of applications (locally installed desktop applications versus remote web application components), but the comparison does illustrate that the techniques applied by a sandbox have an impact on both how widely a particular sandbox can be applied and how it can fail to protect the underlying system.

Software complexity adds an additional wrinkle. Consider Adobe Reader, which has as a robust PDF parser, a PDF renderer, a JavaScript engine, a Digital Rights Management engine, and other complex components critical to the application's function. It may be extremely difficult to find a sandbox with the right combination of techniques to effectively encapsulate Reader without introducing significant complexity in applying the sandbox itself. Even when breaking up these components for sandboxing purposes, the architect must apply a sandbox where the combination of sandboxing techniques is powerful enough to mitigate the threats faced by the component while also applying the techniques in a manner that is acceptable given the unique characteristics and requirements of each sandboxed computation. Additionally, the sandboxed components must be composed in a way where the composition is still secure [36, 49]. If the combination is not secure, it may be possible for an attacker to bypass the sandbox, e.g. by hopping to an unsandboxed component. The complexity may make the sandbox itself a target, creating an avenue for the attacker to compromise the sandbox directly.

In this paper we propose a sandboxing technique referred to as *in-nimbo sandboxing* to address some of the shortcomings of traditional sandboxing techniques. In-nimbo sandboxing leverages low value computing environments to allow defenders greater control over their attack surface, thus channeling attackers to an attack surface an architect can more confidently defend. The low value environment can be located separately from the target system, so an exploit of the sandboxed component (and any additional sandboxes in

the low value environment) has less opportunity to compromise the system we are defending.

Our technique is further motivated in section 2. We discuss an in-nimbo sandbox prototype we built and deployed for Adobe Reader and its performance in section 3. Section 4 compares our in-nimbo sandbox with applying a local sandbox to Reader. We look at thought experiments for applying in-nimbo sandboxing in section 5 before concluding in section 6.

2. IN-NIMBO SANDBOXING

In this section we motivate the need for in-nimbo sandboxing by looking closer at general weaknesses in traditional sandboxes and discuss characteristics of a more suitable environment for executing potentially vulnerable or malicious computations. We then discuss a general model for in-nimbo sandboxing that approximates our ideal environment.

2.1 Why In-Nimbo Sandboxing?

Most mainstream sandboxing techniques are *in-situ*, meaning they impose security policies using only Trusted Computing Bases (TCBs) within the system being defended. In-situ sandboxes are typically retrofitted onto existing software architectures [41, 42, 43, 52, 48, 55, 6] and may be scoped to protect only certain components: those that are believed to be both high-risk and easily isolatable [47, 2]. Existing in-situ sandboxing approaches decrease the risk that a vulnerability will be successfully exploited, because they force the attacker to chain multiple vulnerabilities together [46, 18] or bypass the sandbox. Unfortunately, in practice these techniques still leave a significant attack surface, leading to a number of attacks that succeed in defeating the sandbox. For example, a weakness in Adobe Reader X's sandbox has been leveraged to bypass Data Execution Prevention and Address Space Layout Randomization (ASLR) due to an oversight in the design of the sandbox [20]. Experience suggests that, while in-situ sandboxing techniques can increase the cost of a successful attack, this cost is likely to be accepted by attackers when economic incentives align in favor of perpetrating the attack.[1] The inherent limitation of in-situ techniques is that once the sandbox has been defeated, the attacker is also "in-situ" in the high-value target environment, where he can immediately proceed to achieve his goals.

In order to avoid the inherent limitations of in-situ sandboxing approaches, we propose that improved security may be obtained by isolating vulnerable or malicious computations to *ephemeral computing environments* away from the defended system. Our key insight is that if a vulnerable computation is compromised, the attacker is left in a low-value environment. To achieve his goals, he must still escape the environment, and must do so before the ephemeral environment disappears. The defender controls the means by which the attacker may escape, shaping the overall attack surface to make it more defensible, thereby significantly increasing the cost of attacks compared to in-situ approaches while simultaneously reducing the consequences of successful attacks.

[1]Google Chrome went unexploited at CanSecWest's Pwn2Own contest for three years. Then in 2012, Google put up bounties of $60,000, $40,000, and $20,000 in cash for successful exploits against Chrome. Chrome was successfully exploited three times [25].

We use the term *ephemeral computing environment* to refer to an ideal environment whose existence is short, isolated (i.e. low coupling with the defended environment), and non-persistent, thus making it fitting for executing even malicious computations. A number of environments may approach the ideal of an ephemeral computing environment, for example, Polaris starts Windows XP applications using an application-specific user account that cannot access most of the system's resources [53]. Occasionally deleting the application's account would further limit the scope of a breach. Terra comes even closer by running application specific virtual machines on separate, tamper resistant hardware [24]. Terra requires custom hardware, complicated virtual machine and attestation architectures, and doesn't outsource risk to third parties. In this paper we focus on cloud environments. Cloud computing closely approximates ephemeral environments, as a virtual computing resource in the cloud is isolated from other resources through the combination of virtualization and the use of separate infrastructure for storage, processing, and communication. It may exist just long enough to perform a computation before all results are discarded at the cloud. We call this approach *in-nimbo sandboxing*.

Executing computations in the cloud gives defenders the ability to customize the computing environment in which the computation takes place, making it more difficult to attack. Since cloud environments are ephemeral, it also becomes more difficult for attackers to achieve persistence in their attacks. Even if persistence is achieved, the cloud computing environment will be minimized with only the data and programs necessary to carry out the required computation, and so will likely be of low value to the attacker.[2] In order to escape to the high-value client environment, the attacker must compromise the channel between the client and the cloud. However, the defender has the flexibility to shape the channel's attack surface to make it more defensible.

To make the idea of in-nimbo sandboxing clear, consider Adobe Reader X. Delegating untrusted computations to the cloud is quite attractive for this application, as Reader in general has been a target for attackers over several years. As described more in section 3, we have built and experimentally deployed in an industrial field trial an in-nimbo sandbox for Reader that sends PDFs opened by the user to a virtual machine running in a cloud. An agent on the virtual machine opens the PDF in Reader. Users interact with the instance of Reader that is displaying their PDF in the cloud via the Remote Desktop Protocol (RDP). When the user is done with the document, it is saved back to the user's machine and the virtual machine running in the cloud is torn down. This example illustrates how a defender can significantly reshape and minimize the attack surface.

2.2 Complementary Prior Work

Martignoni et al. have applied cloud computing to sandbox computations within the cloud in an approach that is complementary to ours [37]. Their trust model reverses ours: whereas our model uses a public, low-trust cloud to carry out risky computations on behalf of a trusted client, they use a

[2]There may still be some value to the attacker in the compromised cloud machines, but this is now the cloud provider's problem, which he is paid to manage. This ability to *outsource risk* to the provider is a significant benefit of in-nimbo sandboxing from the point of view of the client.

private, trusted cloud to carry out sensitive computations that the client is is not trusted to perform. They utilize Trusted Platform Modules to attest to the cloud that their client-end terminal is unmodified. They must also isolate the terminal from any malicious computations on the client. Our technique assumes security precautions on the side of the public cloud provider—an assumption we feel is realistic, as cloud providers already have a need to assume this risk.

The scenarios supported by the two techniques are complementary, allowing the application of the two techniques to different components of the same system. For example, Martignoni's sandbox may be used for performing particularly sensitive operations such as online banking, while our technique is useful in the same system for executing untrusted computations from the Internet. These choices reflect the varying trust relationships that are often present in software systems.

2.3 General Model

There are good reasons to reduce the TCB required to execute applications and entire operating systems [38, 40, 39, 50]. The idea is to completely isolate unrelated computations from each other, and to use a TCB that is small enough to be verified, thus reducing and localizing the attack surface to a small, throughly vetted subset of the system's code.

In-nimbo sandboxing addresses this challenge by allowing designers, even when working in legacy environments, flexibility to design TCB(s) to suit their specific context, thus channeling the attacker to a designer-chosen attack surface. This is significantly more flexible as it allows designers to achieve the benefits of a minimal TCB in current commodity hardware/software systems, largely unconstrained by the particular engineering decisions of those systems. When applying in-situ sandboxes, an architect is limited to applying the sandboxing techniques that are supported by the instruction set, operating system, application type, etc., of the system she is trying to defend. These challenges can be particularly difficult to address when vendor software must be sandboxed. However, in-nimbo sandboxes can limit the majority of the attack surface to the communication channel between the client and the cloud. The designer can design a communication channel they are adequately prepared to defend.

In general, in-nimbo sandboxes contain the following:

- A specialized ***transduction mechanism*** in the computing environment we are trying to protect (the principal computing environment) that intercepts invocations of untrusted computations and transfers them to the high value TCB Architecture (see below) on the same system. The transduction mechanism also receives results from the high value TCB architecture and manages their integration into the rest of the system.

- A ***high value TCB architecture*** that sends the untrusted computation and any necessary state to an ephemeral computing asset, separate from the principal computing asset. The high value TCB architecture receives the results of the computation and state from the cloud, verifies both, and transfers them back to the transduction mechanism. We use the term ***TCB architecture*** to reflect the fact that our TCB(s) may be nested in or otherwise cooperate with another TCB (e.g., another sandbox). The nested TCBs can thus compensate for each other's faults and oversights and add redundancy. An attacker must counter each TCB in the architecture to compromise the overall system. In the case of the high value TCB architecture, this could allow an attacker to compromise the system we are trying to defend.

- The cloud executes untrusted computations in a ***low value TCB architecture*** and sends results and state back to the high value TCB architecture.

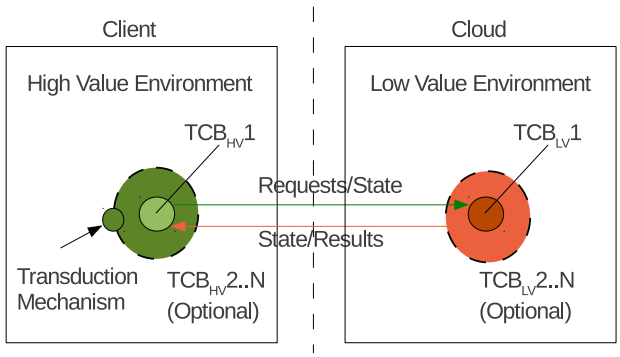

Figure 1: A general model of an in-nimbo sandboxing showing the transduction mechanism and the TCB architecture model. The circles with dashed borders represent one or more TCBs that contain the primary TCBs in each environment. The primary TCB in the high value environment ($TCB_{HV}1$) is responsible for sending requests and state to the low value environment's primary TCB ($TCB_{LV}1$). $TCB_{LV}1$ performs any necessary computations and returns state and results to the $TCB_{HV}1$, which must verify the results. The transduction mechanism moves computations and results into and out of the high value TCB architecture respectively.

The components and their data flows are depicted in figure 1. By picking these components and the contents of the data flows, the defenders effectively channel an antagonist to an attack surface the defenders are confident they can protect. An attacker must bypass or break every TCB in both TCB architectures or bypass the transduction mechanism to successfully compromise the high value environment.

3. CASE STUDY

In-nimbo sandboxing can be applied to sandboxing entire applications or just selected components/computations. In this section we look at the design of an in-nimbo sandbox for Adobe Reader that we prototyped and experimentally deployed at a large aerospace company. We then discuss the basis comparing our sandbox with an in-situ sandbox for Adobe Reader. In the last section, we discuss other uses for in-nimbo sandboxes.

3.1 Why an In-Nimbo Sandbox for Adobe Reader?

Adobe Reader, hereafter referred to simply as Reader, has been subject to numerous attacks since roughly 2008. To make matters worse, its source code was recently stolen [17]. These attacks are unsurprising since Reader has historically been an obvious path of least resistance for actors seeking to execute targeted attacks [27, 10]:

- It has enjoyed a wide installed base in academia, industry, and government.
- Its purpose is to parse and render potentially extremely complex inputs (PDF files) from potentially unknown sources, where the inputs represent active content created in complex and fully capable programming languages such as JavaScript. Version 1.7 of the PDF Reference [7] is 1,310 pages, while the corresponding ISO standard [30] weighs in at 756 pages in length not counting supplements.[3] To complicate matters, there are numerous variations on the standard meant to accommodate the needs of niches such as archiving [29], graphic exchange [28], and several others. This complexity explodes when we consider all of the standards (e.g., font, cryptography, image, active content, movie, audio, and form standards) on which PDF is dependent.
- PDFs are essential and ubiquitous in many places where layouts must be preserved across platforms and environments.
- PDF viewers, including Reader, tend to be written in unmanaged, weakly typed languages (primarily C/C++) that result in applications that are intractable to verify for security attributes.

These points combine to create a target for attack that is likely continue to offer vulnerabilities and is likely available on a multitude of machines worth compromising.

One common suggestion from the security community is to deal with this issue by using alternative PDF viewers [8, 15, 32], which may not suffer from the same vulnerabilities. But in fact many of these share code, because Adobe licenses its PDF parser/renderer [14]. The advice nonetheless has some merit because even if an alternative viewer contains the exact same vulnerability as Reader, an exploit PDF that targets Reader won't necessarily work "out of the box" on an alternative. Many organizations, however, have grown dependent on PDF's more exotic features, such as fly-through 3D models, forms that can be filled out and then digitally signed by a user using hardware tokens, or embedded multimedia in a wide range of formats. At the time of this writing (late-2013), many of these types of features are not supported by many of the alternative viewers.

Let us consider, therefore, an organization with needs that require use of Reader, which has a significant, complex attack surface to defend. We examined the vulnerability streams (essentially RSS feeds of vulnerability reports) provided by NIST's National Vulnerability Database. We only used streams that contained a full years worth of data. The census results in figure 2 show an increasing number of vulnerabilities per year since 2008,[4] with a possible downward trend starting in 2011. There is great diversity in where the vulnerabilities reside in the code, as shown in figure 3. (Sometimes a single vulnerability entry describes multiple vulnerabilities,[5] thus our results undercount how many vulnerabilities have actually been reported.) The vulnerable component distribution in figure 3 was determined by coding all entries by hand that were identified as belonging to Reader and where the coded Common Weakness Enumeration (CWE) implicitly identified a component or a component was explicitly named. As a result, our results likely understate the diversity of vulnerabilities in Reader.

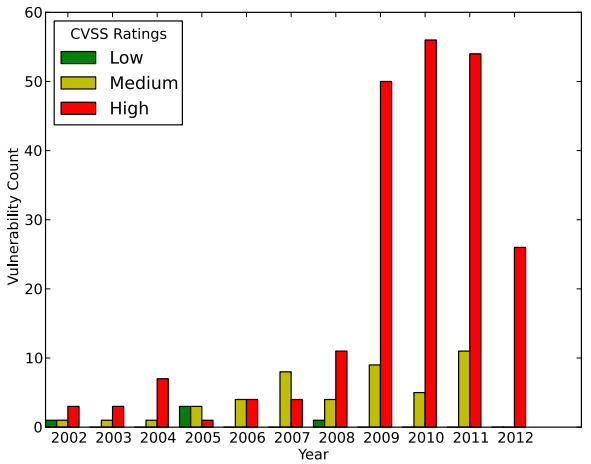

Figure 2: The distribution of vulnerabilities discovered in Reader where data was available for the entire year.

Adobe attempted to address these issues in mid-2010 by sandboxing a subset of Adobe Reader X, and this could be a cause of the decline shown in figure 2. (It is not clear if the apparent slight decrease in reported Reader vulnerabilities is attributable to better application security practices at Adobe, the application of a sandbox, more bundled reports, or some other cause.) As mentioned in section 2, cracks in this sandbox have now been publicly exposed. While the number of vulnerabilities appears to have decreased starting in 2011, many of the vulnerabilities did became more difficult to exploit due to the sandbox.

Others have attempted to detect malicious PDFs [9, 22, 33, 54, 35, 45, 51, 21, 5] with modest success even in the best cases. A systematic analysis [34] substantiates the inadequacy of many techniques for detecting malicious PDFs. We believe that even if detection methods advance, they will never be adequate against advanced attackers. The sheer

[3]Adobe's PDF Reference looks at the file format using Reader's implementation as the point of reference, while the standard just looks at the format on its own without considering a concrete implementation.

[4]The number of vulnerabilities was determined by counting how many vulnerability entries contained a `cpe-lang:fact-ref` field with a name attribute containing the text `adobe:reader`, `adobe:acrobat`, or `adobe:acrobat_reader`.

[5]When a vendor tracks multiple vulnerabilities separately internally but discloses them in one bulletin with too few details to separate them aside from proprietary vulnerability identifiers, NIST tends to report all of the issues under one Common Vulnerability Enumeration identifier.

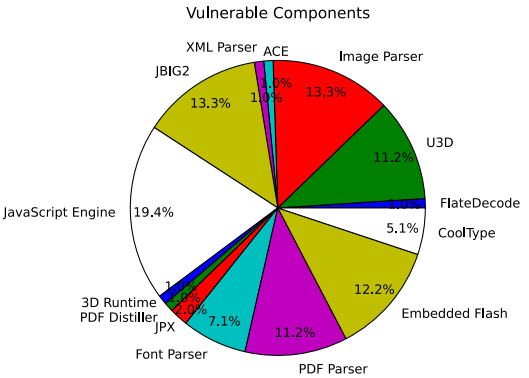

Figure 3: The distribution of vulnerabilities in Reader amongst identifiable components. This distribution is intended to show that problems in Reader are so diverse there is no clear place to concentrate defensive efforts outside of sandboxing.

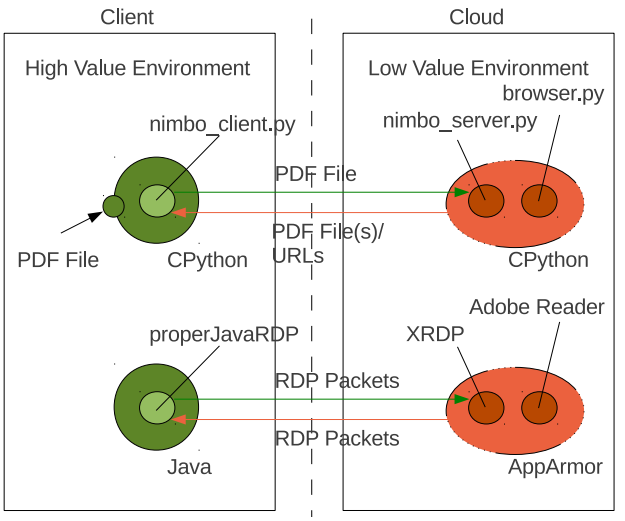

Figure 4: An in-nimbo sandbox for Adobe Reader.

complexity involved in detecting every possible attack in a format as massive as PDF is prohibitive. Additionally, several of Reader's components that have been vulnerable take inputs that involve fully capable programming languages such as JavaScript.

Additional comparison between in-nimbo sandboxing for PDF's and other defensive techniques appears in the appendix. The appendix characterizes several other defensive techniques, defines the criteria used to compare them, and compares the approaches in a criteria matrix.

In-nimbo sandboxing allows Reader to be executed in a low value environment, thus bypassing the complexity concerns detailed in this section in dealing with the richness of the PDF standard. This analysis led us to design and build an in-nimbo sandbox for Adobe Reader.

3.2 Design

To demonstrate the ability of an in-nimbo sandbox to support rich features, we set out with the goal of designing and building a sandbox for Reader that can support a user clicking links in a PDF, filling out and saving forms, interacting with multiple PDFs in the same session, printing PDFs, copying and pasting data to and from a PDF, and interacting with an embedded 3D model. We neglect features such as signing PDF documents with a smart card and several other non-trivial features Adobe advertises (though these features are likely supported via the RDP implementation we used). As an additional convenience, we decided that any Reader settings changed by the user should persist across their sandbox sessions. These design choices ensure the user's interactions with in-nimbo Reader are substantially similar to their typical interactions with in-situ Reader. In fact, aside from a few missing but less commonly used features, the user's interaction only differs in the appearance of the window chroming.

Figure 4 shows a high-level structural model of our prototype in-nimbo sandbox for Adobe Reader. The transduction mechanism consists of a file association between the PDF file extension and nimbo_client.py. This small script and its infrastructure is the primary TCB in the high value environment (24 lines of code). When a PDF file is opened nimbo_client.py transfers the file to nimbo_server.py running in a cloud virtual machine instance. If a sandboxed session does not yet exist, a new session is created. Otherwise, the new PDF is opened as a new tab in the already open instance of Reader.

The user interacts with PDFs opened in the sandboxed version of Reader over an encrypted RDP connection. When Reader, the RDP session, or the RDP client is closed, all of the PDFs in the sandbox are sent back to nimbo_client.py. The PDFs must be returned to the high value client environment because the user may have performed an operation in the sandbox that changed the PDF, such as filling out and saving a form. After the PDFs are returned, nimbo_server.py restores the home directory of the in-nimbo user account that runs Reader to its original state. The cloud environment can always start virtual machines from a clean template, but alternatively resetting the home directory can enable a virtual machine to be re-used, e.g., due to high congestion. The account that runs Reader has limited access to the rest of the system.

When a user clicks a web link in a PDF, that link is sent to the nimbo_client.py and opened on the workstation. If the user does not want links to be opened on their workstation due to the risk of loading a potentially malicious site, they could instead have links opened in an in-nimbo version of their favorite browser. In this way, in-nimbo sandboxes can be composed. Sandbox composition is useful in this case because it prevents any one sandbox from becoming too complicated, and, in particular, having an overly rich attack surface.

3.3 Performance

Our prototype sandbox's performance evaluation is limited by two factors: transfer rates and inefficiencies in the cloud virtual machine set-up for the field trial. But even with inefficiencies in our virtual machine set-up, user perception of performance is comparable with Reader's perfor-

mance locally. Transfer rates dominate the time between when a user opens a PDF and when the user can interact with the PDF, but this rate is typically limited by the connection's upload rate at the workstation. As a result, the upload time statistics presented in this section are not intrinsic to in-nimbo sandboxing and may vary with different connections. The statistics were gathered in our field trial using an in-nimbo sandbox that was deployed as the company would typically deploy cloud applications. The measurements as a whole provide evidence that the approach performs acceptably for typical users.

The Python scripts use the same efficient file transfer schemes used by FTP clients. The server prototype in the cloud implements several optimizations, including queuing virtual machines with Reader already running and waiting for a user to connect. However, the infrastructure software running on the cloud virtual machines is not optimized for our use case. For example, establishing a connection with RDP would be faster if the connection was initialized ahead of time (i.e. subsequent connections via RDP to the virtual machine are much faster than the first connection). This is a side-effect of our choice to use Linux, X server, and xrdp. The issue does not exist on Windows with the RDP server Microsoft provides. It is also possible that xrdp can be modified to remove the issue. Table 1 summarizes results from our industrial collaborator who used an internal cloud and a Microsoft Windows client.

We measured performance with our collaborator by opening a 1 megabyte (MB) PDF ten times. We decided to use a 1 MB PDF after inspecting a corpus of about 200 PDFs characteristic of the use cases of the users and averaging the sizes of its contents. The corpus was collected by an engineer over three years and included the types of PDFs an average engineer encounters throughout their day: brochures, technical reports, manuals, etc. Most PDFs in the corpus were on the order of a few hundred kilobytes, but a small number of the PDFs were tens of MB in size.

For our measurements the high-value environment was 1,800 miles away from the datacenter hosting our cloud. While we parallelize many of the operations required to get to the point where the user can interact with the PDF, the largest unit of work that cannot be comfortably parallelized is transferring the PDF itself. In our tests, the user could interact with our 1 MB PDF within 2.1 seconds, which compares favorably to the 1.5 seconds it takes to interact with a PDF run in Reader locally instead of in-nimbo. The Reader start-up difference is due to the fact that the virtual machine is much lighter than the workstation. The virtual machine doesn't need to run anti-virus, firewalls, intrusion prevention systems, productivity software, and other applications that slow down the workstation.

Though our sandbox increases the startup time, sometimes by several seconds in the case of large PDFs due to the transfer time, we observed no performance issues on standard broadband connections in the United States when interacting with the PDF. The sandbox also performed well when running malicious PDFs that were collected when they were used in attempted attacks targeted at our collaborator's employees. The malware did not escape the sandbox, nor did it persist across sessions. Our results suggest that our technique is currently best applied to longer running, interactive computations unless the running time for the entire initialization process is negligible. The aerospace company we worked with is currently evaluating whether or not to transition the sandbox into production for day-to-day use by high-value targets within the company (e.g. senior executives).

PDF Size	1 MB
Average upload time	2.1 +/- 0.3 seconds*
Average Adobe Reader start time in-nimbo	0.5 seconds
Average time to establish RDP channel	1.5 seconds
Average time until user can interact	2.1 seconds
Distance from client to cloud	1,800 miles
Average Adobe Reader start time in-situ	1.5 seconds

Table 1: Performance metrics for an in-nimbo sandbox using a cloud internal to an enterprise and a Microsoft Windows client. Figures based on 10 runs. The upload time is the primary bottleneck. *Confidence level: 95%

3.4 Limitations

The sandbox prototype does not support the most recent features of PDF because we used the Linux version of Reader, which is stuck at version 9. This limitation is an accidental consequence of our expedient implementation choices and is not intrinsic to in-nimbo sandboxing. It is possible, for example, to instead run Windows with the latest version of Reader in the cloud virtual machine, but this setup would not substantially influence our performance results given the dominance of the transfer rate. (Furthermore, it is possible to run newer Windows versions of Reader in Linux. Adobe Reader X currently has a Gold rating in the WINE AppDB for the latest version of WINE [1]. The AppDB claims that a Gold rating means the application works flawlessly after applying special, application specific configuration to WINE.)

Our malware test of the sandbox is limited by the fact that we didn't have access to malicious PDFs directly targeting Linux or malware that would attempt to break out of our sandbox.

4. IN-NIMBO ADOBE READER VS. IN-SITU ADOBE READER X

In this section we make a structured comparison between our in-nimbo sandbox for Reader with an in-situ Adobe Reader. First, we summarize the framework we'll use for the comparison, and then we apply the framework. The purpose of the framework is to support a systematic exploration of our hypothesis that in-nimbo sandboxing leads to attack surfaces that (1) are smaller and more defensible and (2) offer reduced consequences when successful attacks do occur. The framework is necessarily multi-factorial and qualitative because quantification of attack surfaces and the potential extent of consequences remains elusive.

4.1 Structuring the Comparison

To compare sandboxes we consider what happens when the sandbox holds, is bypassed, or fails. A sandbox is *bypassed* when an attacker can accomplish his goals by jumping from a sandboxed component to an unsandboxed component. A sandbox *fails* when an attacker can accomplish his goals from the encapsulated component by directly attacking the sandbox. The key distinction between a sandbox

bypass and a failure is that any malicious actions in the case of a bypass occur in a component that may have never been constrained to prevent any resulting damage. In a failure scenario, the malicious actions appear to originate from the sandbox itself or the encapsulated component, which creates more detectable noise than the case of a bypass. A bypass can occur when an insufficient security policy is imposed, but a failure requires a software vulnerability. These dimensions help us reason about the consequences of a sandbox break, thus allowing us to argue where in the *consequences* spectrum a particular sandbox falls within a standard risk matrix. To place the sandbox in a risk matrix's *likelihood* spectrum (i.e. the probability of a successful attack given the sandbox), we consider how "verifiable" the sandbox is. Our risk matrix only contains categories (e.g. low, medium, or high) that are meaningful to the comparison at hand. Finally, we rank the outcomes that were enumerated within the argument by their potential hazards, which helps highlight the difference between risk categories.

4.2 Comparing In-Nimbo Adobe Reader to In-Situ Adobe Reader

Adobe Reader X's sandbox applies features of Microsoft Windows to limit the damage that an exploit can do. The Reader application is separated into a low privilege (sandboxed) principal responsible for parsing and rendering PDFs and a user principal responsible for implementing higher privilege services such as writing to the file system. The sandboxed principal is constrained using limited security identifiers, restricted job objects, and a low integrity level. The higher privilege principal is a more stringently vetted proxy to privileged operations. The sandboxed principal can interact with the user principal over a shared-memory channel. The user principal enforces a whitelist-based security policy on any interactions from the sandboxed principal that the system administrator can enhance. Under ideal circumstances the sandboxed principal is still capable of reading secured objects (e.g., files and registry entries),[6] accessing the network, and reading and writing to the clipboard without help from the user principle.

The consequences of an attack on Reader, even when the sandbox holds, are high. In its default state, a PDF-based exploit could still exfiltrate targeted files over the network without any resistance from the sandbox. If the sandbox is successfully bypassed, the attacker can leverage information leakages to also bypass mitigations such as ASLR as mentioned in section 2. Such bypasses, which are publicly documented, are likely to be serious enough to allow a successful attack to install malware on the targeted machine. If other bypass techniques exist, they could allow an attacker to perform any computations the user principal can perform. These outcomes are ranked from most to least damaging in figure 5. Overall, the consequences generally fall into one of the following categories:

- The integrity of the PDF and viewer are compromised
- The confidentiality of the PDF is compromised
- The availability of reader is compromised

[6]Windows Integrity Levels can prevent read up, which stops a process from reading objects with a higher integrity level than the process. Adobe did not exercise this capability when sandboxing Adobe Reader X.

In-Situ Reader X
Install malware on the defended workstation
Perform any computation the user principal can perform
Exfiltrate workstation data on the network
Read files on the workstation filesystem

In-Nimbo Reader
Spy on opened PDFs in the cloud
Abuse cloud resources for other computations

Figure 5: Likely sandbox "consequence outcomes" ranked from most damaging at the top to least damaging at the bottom. Each sandbox's outcomes are independent of the other sandbox's.

- The security (confidentiality, integrity, availability) of the cloud infrastructure is compromised
- The security of the high value environment is compromised

The Reader sandbox is moderately verifiable. It is written in tens of thousand of lines of C that are heavily based on the open-source sandbox created for Google Chrome. The design and source code were manually evaluated by experts from several independent organizations who also implemented a testing regime. According to Adobe, source code analysis increased their confidence in the sandbox as its code was written. The operating system features on which it depends are complex; however, they were implemented by a large software vendor that is known to make use of an extensive Secure Development Lifecycle for developing software [44].

Figure 6 summarizes our qualitatively defined risk for Reader X's sandbox against Reader running in our in-nimbo sandbox. The in-nimbo sandbox has a lower consequence (from the standpoint of the user) because exploits that are successful against Reader may only abuse the cloud instance Reader runs in. The operator of the cloud instance may re-image the instance and take action to prevent further abuse. However, to abuse the cloud instance the attacker would have to both successfully exploit Reader and bypass or break additional sandboxing techniques we apply in the cloud. The exploit must either not crash Reader, or its payload must survive the filesystem restoration and account log-off that would occur if Reader crashed due to the exploit (see 3.1 for details).

The attacker could potentially *bypass* the sandbox by tricking our mechanism into opening the PDF in a different locally installed application capable of rendering PDFs. For example, the attacker may disguise the PDF file as an HTML file, causing it to be opened in the browser if the transduction mechanism is only based on file associations. The browser might have an add-on installed that inspects the document, determines it is a PDF regardless of extension, and then renders the PDF. While this attack would eliminate the benefit of the sandbox, it is not likely to be successful if the user verifies there is not a local PDF viewer installed/enabled (an important part of configuration). The transduction mechanism can also simply use a richer means of determining whether or not a file is a PDF.

The sandbox could *fail* in a way that compromises the user by either an exploitable vulnerability in our 273 line Python-

based TCB (and its infrastructure), the Java RDP client we use, or a kernel mode vulnerability exploitable from either. Such a failure would require first successfully compromising the cloud instance as discussed earlier and then finding an interesting vulnerability in our small, typesafe components. In other words, a failure of the sandbox requires that the TCBs in both the client and the cloud fail.

Another potential point of concern is that the cloud instance's hypervisor could be compromised, thus compromising other virtual machines managed by that hypervisor or even the entire cloud. We do not consider this issue in our analysis because our sandbox is a use of the cloud, not an implementation of a cloud. One of the key selling points behind using a cloud environment is that the provider manages the infrastructure; they take on the risk and management expenses. The ability to outsource risk that cannot be eliminated to a party that is willing to assume the risk is a key advantage of our approach. Additionally, our technique does not add a new threat to clouds in the sense that anyone can rent access to any public cloud for a small sum of money and attempt to compromise the hypervisor. Finally, we are primarily sandboxing computations because we don't trust them. In the case of the Reader sandbox, a compromise could cause sensitive PDFs to be stolen, which would still be better than the compromise of an entire workstation full of sensitive information.

In short, the in-nimbo sandbox is easier to verify and requires more work for an attacker to achieve enough access to compromise the client. Adobe Reader X's sandbox is harder to verify and allows the workstation it is running on to be compromised even if the sandbox holds. Due to the well known characteristics of each sandbox, we consider this evaluation to be reasonable evidence of the validity of our hypotheses that in-nimbo sandboxing leads to smaller, more defensible attack surfaces and reduced consequences in the event of successful attacks. While we only evaluated one sandbox in addition to our in-nimbo sandbox and for only one application, the results of the evaluation are largely influenced by issues that are fundamental to in-situ sandboxing when compared to in-nimbo sandboxing.

Consequence			
High		In-Situ Reader X	
Low	In-Nimbo Reader		
	Easy	*Moderate*	
	Likelihood (Verifiability)		

Figure 6: A grid summarizing our evaluation of Reader X and our In-Nimbo sandbox for Reader.

5. IN-NIMBO THOUGHT EXPERIMENTS

In this section we consider the potential of applying the in-nimbo sandboxing technique for a subset of HTML5 and for protecting proprietary data. There are other examples (not elaborated here) that would be similar to our sandbox for Reader, such as an in-nimbo sandbox for Microsoft Word or Outlook.

5.1 Selective Sandboxing

HTML5 specifies a canvas element [13] that provides scripts with a raster-based surface for dynamically generating graphics. By default, browsers must serialize the image to a PNG for presentation to the user (other raster formats may be

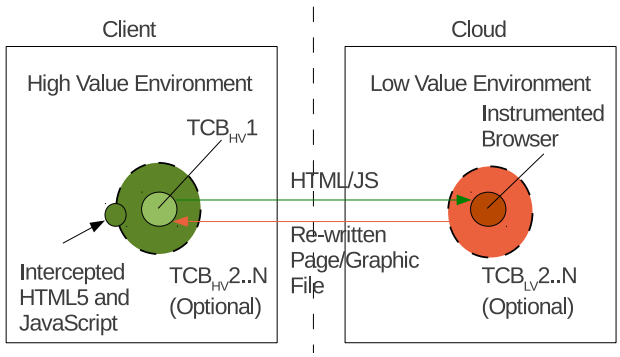

Figure 7: The model for an in-nimbo sandbox for HTML5 canvases.

specified). For an animated surface, the serialized image represents the currently visible frame. Unfortunately, early implementations of such rich browser features have continually been prone to vulnerabilities. CVE-2010-3019 documents a heap-based buffer overflow in Opera 10's implementation of HTML5 canvas transformations [3]. A stack-based overflow was also recently discovered in Mozilla Firefox's HTML5 canvas implementation[4]. As a result, a risk-focused user may desire that HTML5 canvas computations be sandboxed.

Figure 7 shows the model of an HTML5 canvas in-nimbo sandbox. In this case, the transduction mechanism is a proxy running between a browser and the Internet. The transduction mechanism intercepts and inspects HTML pages for canvas declarations. When a canvas declaration is detected, the proxy collects all referenced JavaScript code and sends the page and scripts to the high value TCB architecture (the client). The client sends the collected HTML and JavaScript to the cloud instance, which utilizes an instrumented browser to render any drawing on the canvas. The canvas is replaced with the image file the browser generates (per the HTML5 standard) when the rendering script finishes. When a loop in the rendering script is detected (i.e. an animation is present), the canvas declaration is replaced with an image tag pointing to a 3 second animated GIF composed of all of the frames the script rendered in that time period. All JavaScript that drew on the canvas is removed, and the cloud returns the re-written JavaScript, HTML, and PNG to the client. The client verifies the image file and checks that no unexpected code/markup changes have been made before sending the results back to the proxy. The proxy passes the modified results to the browser.

This sandbox would effectively confine exploits on HTML5 canvas implementations to our low value computing environment. Furthermore, it would reduce the verification problem from verifying the full HTML5 canvas implementation and its dependencies to that of verifying raster image formats supported by the canvas tag and ensuring that no code has been added to the intercepted files (i.e., code has only been removed and/or canvas tags have been replaced with image tags). While the sandbox does not support animated canvases longer than 3 seconds or whose visual representation is dependent on real-time user input, a user who cannot accept such limitations can use a browser that is fully sandboxed in-nimbo such as Reader was in the previous section. It is also possible that an alternative implementation of this

sandbox could support longer animations and user input.

5.2 Protecting Proprietary Algorithms

Modern audio players allow users to manage libraries of tens of thousands of songs and automatically perform experience-enhancing operations such as fetching album covers from the Internet. More advanced players also attempt to identify songs and automatically add or fix any missing or corrupt metadata, a process known as auto-tagging. Unfortunately, the act of robustly parsing tags to identify an album and artist to fetch a cover is potentially error prone and complicated. CVE-2011-2949 and CVE-2010-2937 document samples of ID3 parsing vulnerabilities in two popular media players [16, 11]. Furthermore, an audio player vendor may consider all steps of the waveform analysis they use to identify untagged audio files to be proprietary. To address these concerns, the vendor may wish to design their application to make use of an in-nimbo sandbox to perform these operations.

Figure 8 shows the model of a possible in-nimbo sandbox for fetching album covers and performing auto-tagging. The transduction mechanism is the audio player itself. When the player detects an audio file that is new it sends the file to the high value TCB architecture (the client). The client sends the audio file to the cloud instance, which performs the task of automatically adding any missing tags to the audio file and fetching the correct album cover. The cloud sends the tagged audio file and album cover to the client, where it will be verified that only the audio files tags have changed, that they comply with the correct tagging standard, and that the album cover is of the expected format and well formed. The client will then send the verified audio file and album cover to the audio player, which will place both into their correct places in the library.

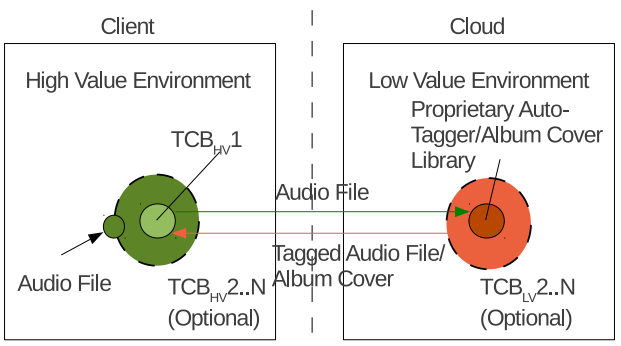

Figure 8: The model for an in-nimbo sandbox for an auto-tagging audio player.

Assuming that all managed audio files make their way through the in-nimbo sandbox at least once, this sandbox effectively mitigates the risk of robustly parsing tags while also not exposing the inner-workings of the proprietary waveform algorithm. Potential limitations are curtailed by intentionally designing the application to utilize an in-nimbo sandbox.

6. CONCLUSIONS AND FUTURE WORK

In this paper we argued that we can improve system security (confidentiality, integrity, availability) by moving untrusted computations away from environments we want to defend. We did so by first introducing one approach for achieving that idea, a category of sandboxing techniques we refer to as in-nimbo sandboxing. In-nimbo sandboxes leverage cloud computing environments to perform potentially vulnerable or malicious computations away from the environment that is being defended. Cloud computing environments have the benefit of being approximately ephemeral, thus malicious outcomes do not persist across sandboxing sessions. We believe this class of sandboxing techniques is valuable in a number of cases where classic, in-situ sandboxes do not yet adequately isolate a computation.

We argued that in-situ sandboxing does not adequately reduce risk for Adobe Reader, thus motivating us to build an in-nimbo sandbox for Reader. We then discussed the design of an in-nimbo sandbox for Reader and presented a structural argument based on five evaluation criteria that suggests that it is more secure and that, with respect to performance, has a user experience latency subjectively similar to that of Reader when run locally. After arguing that our sandbox is more secure than an in-situ sandbox for Reader, we looked at how in-nimbo sandboxes might be built for a couple of other applications that represent different in-nimbo use cases.

Our argument for why our sandbox is better is structured but necessarily qualitative. We believe that many security dimensions cannot now be feasibly quantified. We nonetheless suggest that structured criteria-based reasoning building on familiar security-focused risk calculus can lead to solid conclusions. Indeed, we feel the approach is an important intermediate step towards the ideal of quantified and proof-based approaches.

7. ACKNOWLEDGEMENT

This material is based upon work supported by the Army Research Office under Award No. W911NF-09-1-0273 and by the Air Force Research Laboratory under Award No. FA87501220139.

8. REFERENCES

[1] AppDB Adobe Reader. http://goo.gl/Fx9pd.
[2] Chromium sandbox. http://www.chromium.org/developers/design-documents/sandbox/.
[3] National vulnerability database (NVD) national vulnerability database (CVE-2010-3019). http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2010-3019.
[4] National vulnerability database (NVD) national vulnerability database (CVE-2013-0768). http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-0768.
[5] PDF x-RAY. https://github.com/9b/pdfxray_public.
[6] What is protected view? - word - office.com. http://office.microsoft.com/en-us/word-help/what-is-protected-view-HA010355931.aspx.
[7] *PDF Reference*, sixth edition ed. Adobe Systems Incorporated, Nov. 2006.
[8] Two new vulnerabilities in Adobe Acrobat Reader. http://www.f-secure.com/weblog/archives/00001671.html, Apr. 2009.
[9] Anatomy of a malicious PDF file. http://goo.gl/VlLmU, Feb. 2010.

[10] Military targets. http://www.f-secure.com/weblog/archives/00002203.html, July 2011.

[11] National vulnerability database (NVD) national vulnerability database (CVE-2011-2949). http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-2949, Oct. 2011.

[12] Understanding and working in protected mode internet explorer. http://msdn.microsoft.com/en-us/library/bb250462(v=vs.85).aspx, Feb. 2011.

[13] 4.8.11 the canvas element - HTML5. http://www.w3.org/TR/html5/the-canvas-element.html#the-canvas-element, Mar. 2012.

[14] Adobe PDF library SDK | adobe developer connection. http://www.adobe.com/devnet/pdf/library.html, Aug. 2012.

[15] Google warns of using adobe reader - particularly on linux. http://www.h-online.com/security/news/item/Google-warns-of-using-Adobe-Reader-particularly-on-Linux-1668153.html, Aug. 2012.

[16] National vulnerability database (NVD) national vulnerability database (CVE-2010-2937). http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2010-2937, Jan. 2012.

[17] Arkin, Brad. Illegal access to Adobe source code. http://blogs.adobe.com/asset/2013/10/illegal-access-to-adobe-source-code.html, Oct. 2013.

[18] Buchanan, K., Evans, C., Reis, C., and Sepez, T. Chromium blog: A tale of two pwnies (Part 2). http://blog.chromium.org/2012/06/tale-of-two-pwnies-part-2.html, June 2012.

[19] Clarkson, M. R., and Schneider, F. B. Hyperproperties. *Journal of Computer Security 18*, 6 (2010), 1157–1210.

[20] Delugre, G. Bypassing ASLR and DEP on Adobe Reader X - Sogeti ESEC Lab. http://esec-lab.sogeti.com/post/Bypassing-ASLR-and-DEP-on-Adobe-Reader-X, June 2012.

[21] Esparza, J. peepdf - PDF analysis and creation/modification tool. http://code.google.com/p/peepdf/.

[22] Fratantonio, Y., Kruegel, C., and Vigna, G. Shellzer: a tool for the dynamic analysis of malicious shellcode. In *Proceedings of the 14th international conference on Recent Advances in Intrusion Detection* (Berlin, Heidelberg, 2011), RAID'11, Springer-Verlag, pp. 61–80.

[23] Friedl, S. Best practices for UNIX chroot() operations. http://www.unixwiz.net/techtips/chroot-practices.html.

[24] Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., and Boneh, D. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2003), SOSP '03, ACM, pp. 193–206.

[25] Goodin, D. At hacking contest, Google Chrome falls to third zero-day attack (Updated). http://arstechnica.com/business/news/2012/03/googles-chrome-browser-on-friday.ars, Mar. 2012.

[26] Hamlen, K. W., Morrisett, G., and Schneider, F. B. Computability classes for enforcement mechanisms. *ACM Transactions on Programming Languages and Systems (TOPLAS) 28*, 1 (2006), 175–205.

[27] Higgins, K. Spear-phishing attacks out of China targeted source code, intellectual property. http://goo.gl/8RzyT, Jan. 2010.

[28] International Organization for Standardization, I. ISO 15929:2002 - International Organization for Standardization. http://goo.gl/SUP1A.

[29] International Organization for Standardization, I. ISO 19005-2:2011 - International Organization for Standardization. http://goo.gl/mtHWw.

[30] International Organization for Standardization, I. ISO 32000-1:2008 - International Organization for Standardization.

[31] Jana, S., Porter, D. E., and Shmatikov, V. TxBox: Building secure, efficient sandboxes with system transactions. In *2011 IEEE Symposium on Security and Privacy (SP)* (May 2011), IEEE, pp. 329–344.

[32] Landesman, M. Free PDF readers: Alternatives to Adobe Reader and Acrobat. http://antivirus.about.com/od/securitytips/tp/Free-Pdf-Readers-Alternatives-To-Adobe-Reader-Acrobat.htm.

[33] Laskov, P., and Srndic, N. Static detection of malicious JavaScript-bearing PDF documents. In *Proceedings of the 27th Annual Computer Security Applications Conference* (New York, NY, USA, 2011), ACSAC '11, ACM, pp. 373–382.

[34] Maiorca, D., Corona, I., and Giacinto, G. Looking at the bag is not enough to find the bomb: An evasion of structural methods for malicious PDF files detection. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security* (New York, NY, USA, 2013), ASIA CCS '13, ACM, pp. 119–130.

[35] Maiorca, D., Giacinto, G., and Corona, I. A pattern recognition system for malicious PDF files detection. In *Proceedings of the 8th International Conference on Machine Learning and Data Mining in Pattern Recognition* (Berlin, Heidelberg, 2012), MLDM'12, Springer-Verlag, pp. 510–524.

[36] Mantel, H. On the composition of secure systems. In *Proceedings of the IEEE Symposium on Security and Privacy, 2002* (2002), IEEE, pp. 88–101.

[37] Martignoni, L., Poosankam, P., Zaharia, M., Han, J., McCamant, S., Song, D., Paxson, V., Perrig, A., Shenker, S., and Stoica, I. Cloud Terminal: Secure access to sensitive applications from untrusted systems. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference* (Berkeley, CA, USA, 2012), USENIX ATC'12, USENIX Association, pp. 14–14.

[38] McCune, J. M., Li, Y., Qu, N., Zhou, Z., Datta, A., Gligor, V., and Perrig, A. TrustVisor: Efficient TCB reduction and attestation. In *IEEE Symposium on Security and Privacy (SP), 2010* (2010), IEEE, pp. 143–158.

[39] McCune, J. M., Parno, B., Perrig, A., Reiter, M. K., and Seshadri, A. How low can you go?: Recommendations for hardware-supported minimal TCB code execution. *SIGOPS Oper. Syst. Rev. 42*, 2 (Mar. 2008), 14–25.

[40] McCune, J. M., Parno, B. J., Perrig, A., Reiter, M. K., and Isozaki, H. Flicker: An execution infrastructure for TCB minimization. *SIGOPS Oper. Syst. Rev. 42*, 4 (Apr. 2008), 315–328.

[41] McQuarrie, L., Mehra, A., Mishra, S., Randolph, K., and Rogers, B. Inside Adobe Reader Protected Mode - part 1 - design. http://blogs.adobe.com/asset/2010/10/inside-adobe-reader-protected-mode-part-1-design.html, Oct. 2010.

[42] McQuarrie, L., Mehra, A., Mishra, S., Randolph, K., and Rogers, B. Inside adobe reader protected mode - part 2 - the sandbox process. http://blogs.adobe.com/asset/2010/10/inside-adobe-reader-protected-mode-%E2%80%93-part-2-%E2%80%93-the-sandbox-process.html, Oct. 2010.

[43] McQuarrie, L., Mehra, A., Mishra, S., Randolph, K., and Rogers, B. Inside adobe reader protected mode - part 3 - broker process, policies, and inter-process communication. http://blogs.adobe.com/asset/2010/11/inside-adobe-reader-protected-mode-part-3-broker-process-policies-and-inter-process-communication.html, Nov. 2010.

[44] Michael Howard, and Steve Lipner. *The Security Development Lifecycle*. Microsoft Press, May 2006.

[45] Nedim Srndic, and Pavel Laskov. Detection of malicious PDF files based on hierarchical document structure. In *Network and Distributed System Security Symposium* (2013).

[46] Obes, J., and Schuh, J. Chromium blog: A tale of two pwnies (Part 1). http://blog.chromium.org/2012/05/tale-of-two-pwnies-part-1.html, May 2012.

[47] Sabanal, P., and Yason, M. Playing in the Reader X sandbox. *Black Hat USA Briefings* (July 2011).

[48] Schuh, J. Chromium blog: The road to safer, more stable, and flashier flash. http://blog.chromium.org/2012/08/the-road-to-safer-more-stable-and.html, Aug. 2012.

[49] Sewell, P., and Vitek, J. Secure composition of insecure components. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop, 1999.* (1999), IEEE, pp. 136–150.

[50] Singaravelu, L., Pu, C., Hartig, H., and Helmuth, C. Reducing TCB complexity for security-sensitive applications: three case studies. *SIGOPS Oper. Syst. Rev. 40*, 4 (Apr. 2006), 161–174.

[51] Smutz, C., and Stavrou, A. Malicious PDF detection using metadata and structural features. In *Proceedings of the 28th Annual Computer Security Applications Conference* (New York, NY, USA, 2012), ACSAC '12, ACM, pp. 239–248.

[52] Stender, S. Inside adobe reader protected mode - part 4 - the challenge of sandboxing. http://blogs.adobe.com/asset/2010/11/inside-adobe-reader-protected-mode-part-4-the-challenge-of-sandboxing.html, Nov. 2010.

[53] Stiegler, M., Karp, A. H., Yee, K.-P., Close, T., and Miller, M. S. Polaris: Virus-safe computing for windows XP. *Commun. ACM 49*, 9 (Sept. 2006), 83–88.

[54] Tzermias, Z., Sykiotakis, G., Polychronakis, M., and Markatos, E. P. Combining static and dynamic analysis for the detection of malicious documents. In *Proceedings of the Fourth European Workshop on System Security* (New York, NY, USA, 2011), EUROSEC '11, ACM, pp. 4:1–4:6.

[55] Uhley, P., and Gwalani, R. Inside flash player protected mode for firefox. http://blogs.adobe.com/asset/2012/06/inside-flash-player-protected-mode-for-firefox.html, June 2012.

[56] Wahbe, R., Lucco, S., Anderson, T. E., and Graham, S. L. Efficient software-based fault isolation. *SIGOPS Oper. Syst. Rev. 27*, 5 (Dec. 1993), 203–216.

[57] Yee, B., Sehr, D., Dardyk, G., Chen, J. B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., and Fullagar, N. Native client: a sandbox for portable, untrusted x86 native code. *Commun. ACM 53*, 1 (Jan. 2010), 91–99.

APPENDIX

A. PDF DEFENSE CONCEPT RANKINGS

Alternative mitigations for PDF. Other possible mitigations, such as transforming a PDF to a more defensible format, are important to consider. As noted, these must contend with the challenges of supporting some of the exotic features mentioned in section 3.1 if they are to be usable by the widest possible audience. Table 2 summarizes the main criteria that we believe a more optimal defense concept for PDF must manifest. Table 3 shows a criteria matrix with some evaluation points to assist in comparing format transformation to an in-nimbo sandbox. We compared more than two approaches for defending PDF, but this is sufficient to illustrate the criteria-based approach.

Table 2: A summary of the criteria a more optimal PDF defense might manifest.

Criteria	Description
Simple File Transduction	The PDF must be able to cross between a native and either transformed or sandboxed contexts. For example, in local sandboxing a PDF must be intercepted and placed in the sandbox's file system. There must be a simple and efficient mechanism for performing this operation.
Native State Persistence	Modern readers may persist state in the form of the last page viewed, digital signatures on subsets of the file, and form content. This state should be maintained as it would be if the PDF was interacted with in a traditionally installed fully featured reader.
Advanced PDF Feature Support	PDFs can be used to interchange Flash content, movies, audio, and 3D models to name few examples of advanced content. Additionally, segments of a PDF can be digitally signed, locked for printing, etc. Maintaining support for many of these features (not necessarily all) is paramount.
Low Breakout Risk	There should be little chance that an exploit succeeds in compromising a target workstation.
High Breakout Recovery	If an exploit breaks the defense concept, recovery should be trivial.
Low Performance Overhead	The defense concept should not unacceptably harm performance.
Low Adoption Overhead	Defense concepts should be easy for individuals and enterprises to effectively place into operation.

Table 3: A criteria matrix that compares Format Transformation and In-Nimbo Sandboxing in defending PDF to an unsandboxed version of Reader.

	Format Trans.	In-Nimbo Sandbox
Simple File Transduction	Minimal Change	Degraded
Native State Persistence	Much Degraded	Improved
Adv. PDF Support	Much Degraded	Minimal Change
Low Breakout Risk	Improved	Much Improved
High Breakout Recovery	Minimal Change	Improved
Printing Support	Minimal Change	Minimal Change
Low Performance Ovrhd	Minimal Change	Minimal Change
Low Adoption Ovrhd	Minimal Change	Minimal Change