Making the Unsafe Safer: Zero-Trust Web Access Using Remote Browser Isolation

GIAC (GCIH) Gold Certification

Author: Craig Meyer, camsecure@fastmail.com
Advisor: Michael Long

Accepted: 05/25/2021

Abstract

This paper explores the potential of Remote Browser Isolation (RBI) technology configured as a reverse proxy to protect organizational web applications from untrusted clients. RBI technology is marketed to protect client browsers from compromise when browsing the unsafe Internet. RBI technology may provide additional security protections to web applications accessible through zero-client browser-based remote access in a zero-trust architecture. This paper uses research to conclude that RBI technology would present practical mitigations to many common web application vulnerabilities that can be exploited by an authenticated session on an untrusted client or network. Since these mitigations provide a middle ground between outright blocking or allowing native access, RBI could serve a useful purpose in a zero-trust architecture that must continue to operate for organizational purposes despite security risks.

1. Introduction

This paper is intended for a technically savvy reader focused on solving web application security and zero-trust remote access architecture problems.

In 2020, web applications were involved in over 43 percent of breaches in the previous year (2020 Data Breach Investigations Report, 2020). Therefore, web application security, especially for perimeter-facing applications, is a top priority for security practitioners.

It is common for organizations to set up Internet-facing web applications as a form of clientless remote access for users that can only reach organizational information resources via an unmanaged device. Access to some web applications is more than a convenience service. It is often necessary to provide users access to organizational web applications as a necessity to achieve an organizational mission. The web applications that are exposed in this method are often intended for internal use by trusted machines and users. It is possible to mitigate the risk of untrusted users by enforcing multifactor authentication. However, the risk of an untrusted device is harder to address since the only option available to handle an untrusted device is to block it outright. Unfortunately, this defeats the purpose because zero-client remote access solutions are intended to be accessed from devices that aren't in the organization's control and, therefore, will always present risk. Risks from an unmanaged client may include an attack against a web application or sensitive data leakage to a client not authorized to store and process such data.

A new security technology concept intended to protect clients from web applications arrived in the commercialized security industry called Remote Browser Isolation or RBI. RBI is marketed for use as a forward proxy to protect organizational clients from risks on the Internet while web browsing. This paper explores the idea of using RBI technology as a reverse proxy to protect organizational web applications from untrusted clients as part of a comprehensive zero-client remote access solution. The paper attempts to demonstrate that reverse proxy is a legitimate use case for RBI technology that could significantly reduce risks of web application exposure in a zero-trust architecture.

Craig Meyer, camsecure@fastmail.com

2. Client-Based Remote Access

The traditional method for accessing private information resources from a remote site has been the client-based VPN. Client-based VPNs allow an installed piece of software on the client machine to build a secure and encrypted tunnel into an organization's network. However, client-based VPN solutions have never been sufficient for all remote access use cases, and their limitations are under increasing pressure. One example of a use case that isn't compatible with a client-based VPN is when an employee must spend time at a customer's site where they may not have access to a workstation issued by their organization with the proper VPN client and security controls. This use case often impacts organizations that contract out employees as embedded resources to other organizations. Employees in this situation must retain access to their employer's resources for activities such as reviewing the company intranet site, recording time worked, or tracking project status. An example of increasing pressure on client VPNs is the cultural demand for more access from bring-your-own-device (BYOD) proponents that support the use of personal devices for some subset of organizational activities. BYOD proponents argue that it is inefficient to carry an organizationally provided device when they already have a working personal device. Some organizations also see BYOD as a way to save money by not providing devices to employees or paying for extra cellular plans. These and other motivators have forced organizations to develop and adopt clientless remote access solutions.

2.1. Clientless Web Access

The most common clientless remote access solutions involve making web applications available directly over the Internet. Many organizations do this with a centralized solution that provides numerous capabilities on top of simply granting access, such as:

1. Providing universal multifactor authentication
2. Centralizing and standardizing encryption

3. Performing device posture assessments
4. Creating a choke point for improved network inspection such as through an Intrusion Detection System (IDS) or a Web Application Firewall (WAF)
5. Providing single-sign-on (SSO) for access to multiple applications

One of the issues that organizations deploying a clientless remote access solution must grapple with is that allowing unmanaged devices with unknown security postures to have access to internal web applications can pose a great risk to the applications and the data they store. Some examples of those risks include session cookie theft, machine-in-the-middle attacks, man-in-the-browser attacks, and data leakage issues. All of these risks stem from an architecture that allows an untrusted and potentially unsafe device to directly and programmatically interact with a web application that may not have been designed and secured for public exposure.

2.2. The Challenge of Untrusted Clients

Some organizations attempt to solve these security concerns through posture assessing devices before determining whether to grant or deny access. Posture assessment can be an effective security control, but in practice, it tends to require denying most access. Posture assessment of a device typically requires an installed client on the host to conduct posture checks such as reviewing installed programs, operating system versions, patch levels, and configuration settings. However, since the inability to deploy a client is a chief justification for a clientless option in the first place, many devices simply can't be assessed. When a device fails assessment or can't be assessed at all, it leaves the user of the client no alternative to meet the needs of their use case, which may not be an acceptable outcome for the organization.

Even if a device passes posture assessment in a zero-trust architecture, that doesn't necessarily mean granting full access is necessary. According to the *National Institute of Standards and Technology special publication 800-207 on Zero Trust Architecture*, one of the central tenants of zero-trust architecture is that "access should be granted with the least privileges necessary to complete a task" (S. Rose et al., 2020). Solutions that posture assess web clients before granting access only provide a binary

option to grant full access or block access completely. Typical clientless web remote access solutions offer no ability to reduce risk through limiting privileges per the NIST recommendation. A capability is needed that can offer an in-between solution that reduces risk without eliminating access.

2.3. Remote Browser Isolation as a Potential Solution

Over the last decade, a new technology called Remote Browser Isolation (RBI) has started making waves in the endpoint security market. According to MacDonald (2018), "Remote browser offerings are a subset of browser isolation technologies that remove the browsing process from the end user's desktop and move it to a browser server or cloud-based browser service." RBI technology works by placing the browser rendering engine for a web application on a remote machine, typically in a remote datacenter, and then backhauling a disarmed version of the website back to the actual web client. Remington (2020) described the client browser, in turn, capturing keystrokes, scroll commands, and mouse commands and sending them back to the RBI engine. RBI, therefore, allows the client browser to remote control the virtual RBI browser.

Understanding how Remote Browser isolation works starts with understanding how webpage rendering works under normal circumstances in a browser. Webpage rendering starts when a browser receives an HTML transmission with elements to render. The elements are used to develop a Document Object Model (DOM) tree and a Cascading Style Sheets Object Model (CSSOM) tree. The DOM tree is a breakdown of webpage elements including HTML objects, image objects, and JavaScript objects. The CSSOM tree overlays attributes intended to be applied to the DOM elements to format the final page into a style. After the browser constructs the DOM and CSSOM trees, it combines them into the render tree. The render tree is used to create a layout that determines where the elements will be placed on the screen. Finally, a painting phase takes place, which makes bitmaps that are sent to the screen. The following graphic, based on Hiwarale (2019) detail, provides an overview of the process in steps.

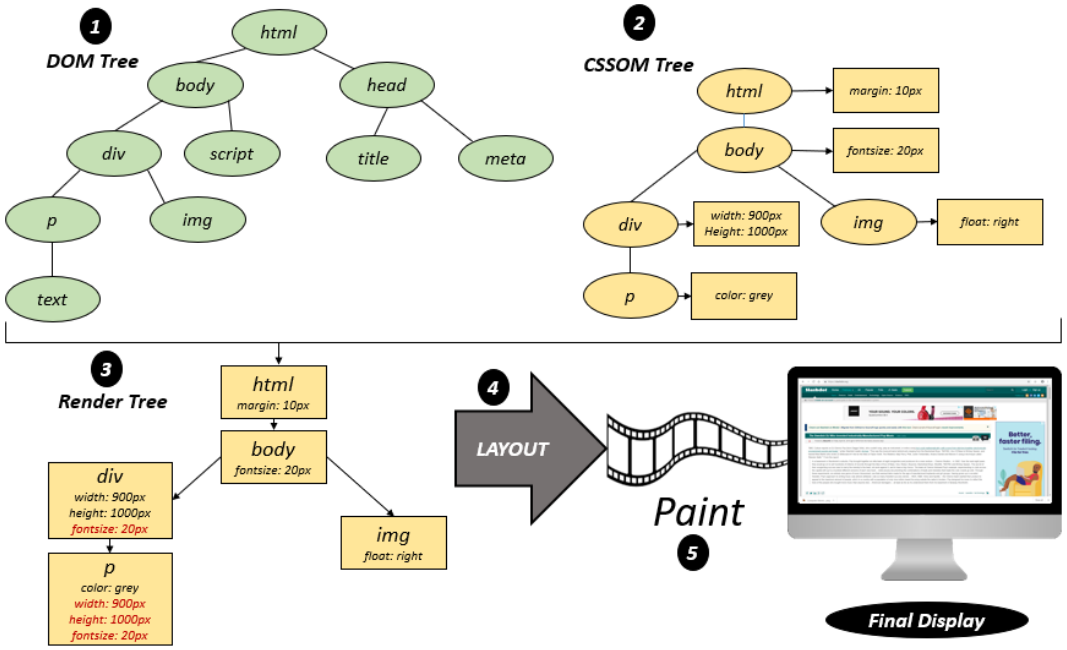


Figure 1 - Steps for Browser Rendering of HTML

There are two RBI methods for sending the remotely rendered and disarmed version of a website to a client browser. The first is often referred to as "pixel-pushing" and involves using a WebSocket to deliver bitmap images that appear identical to the original site. Pixel-pushing is the safer option from a security standpoint as it creates the most significant amount of isolation. However, some limitations include high bandwidth utilization, high CPU utilization, and the loss of some typical browsing features such as text search and clipboard functions. Figure 2 depicts how this pixel-pushing method works in a simple diagram.

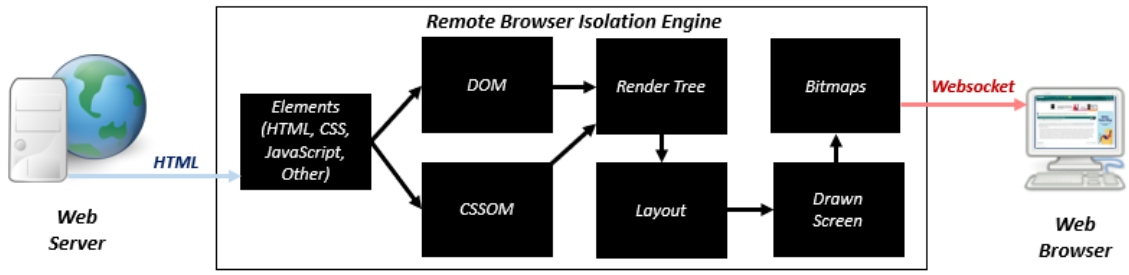


Figure 2 – Pixel-Pushing RBI

Craig Meyer, camsecure@fastmail.com

The second method for sending a remotely rendered version of a website is often referred to as DOM tree mirroring. With DOM tree mirroring, the RBI engine does most of the work to render the page, but rather than moving to the layout and painting phase, the RBI instead performs a DOM translation to create a new but disarmed HTML version of the webpage that can be sent to the client browser for rendering. In translating or mirroring the page, the RBI engine creates a new page that looks like the original page but does not contain the elements of the original content that could be dangerous for the client. From a security perspective, DOM tree mirroring can be more hazardous for the client because some original content still gets through. However, this RBI method has several advantages, including reduced bandwidth usage and the fact that elements of the page presented to the user retain their original form. For example, with DOM tree mirroring, an image can still be saved to disk, or text can be copied to the clipboard. Figure 3 below depicts how the DOM tree mirroring method works.

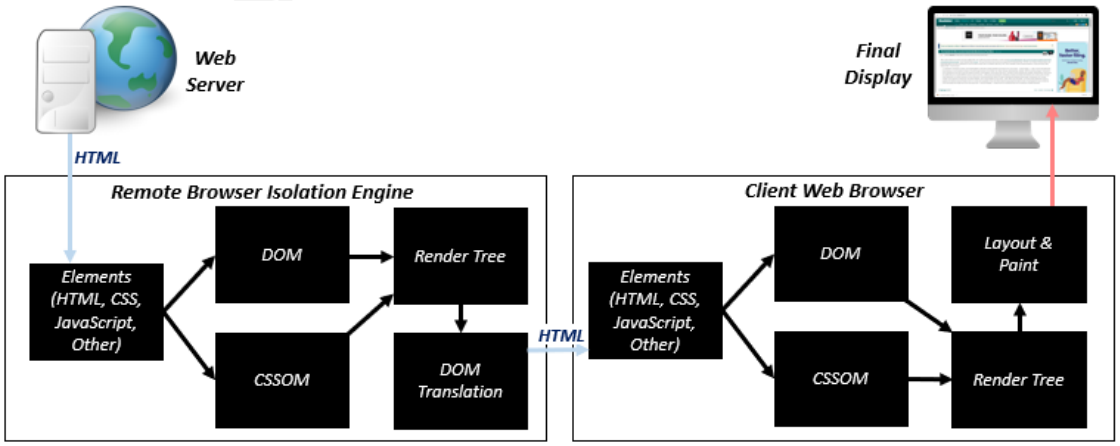


Figure 3 - DOM Tree Mirroring RBI

RBI technology appears to be marketed exclusively for use as a forward proxy to protect an organization's clients from dangers posed by browsing the Internet. RBI is a creative alternative to traditional methods of securing client browser activity such as using URL content filtering, Intrusion Prevention Systems, and signature-based client anti-malware software that regularly fails to prevent attacks. Why try to block and detect attacks on machines when one can simply remove the exposure to them altogether?

Thinking about RBI technology alongside the dangers of clientless remote access to organizational web applications led to the hypothesis that this technology, when used as a reverse proxy, could protect web applications from clients as well. Research was performed to test this hypothesis that included building a lab and testing whether this technology could protect web applications in a novel way.

3. Lab Environment

To test the use of an RBI proxy to block web application attacks originating from an insecure client, a virtual workstation running Ubuntu was deployed using VMware. Chromium was installed on the workstation as a client browser for testing. Burp Suite Community Edition was installed to intercept and modify browsing traffic. Webgoat was installed to test web application attacks. Webgoat is a purposely insecure web application by OWSAP to train professionals on web hacking tactics. Finally, an open-source RBI solution called ViewFinderJS was installed to test isolation. Below is a simple diagram depicting the lab environment on the Ubuntu workstation.

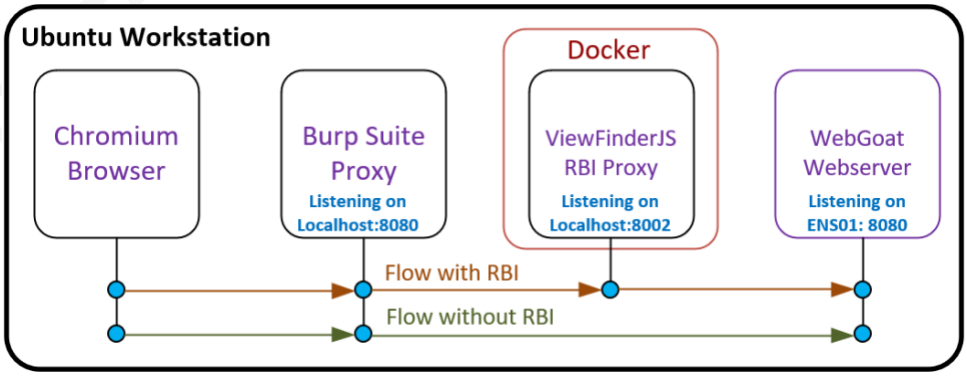


Figure 4 - Lab Design

The following sections further detail the inventory of components and how they were configured and used for testing.

3.1. Ubuntu Operating System

Version 20.04.2 of the Ubuntu Desktop operating system (OS) was used as the base to deploy and execute all other components. Using a working desktop made it possible to perform analysis in a browser, record screenshots of results, and document

notes in a text editor. Ubuntu package management (apt) made it possible to install and update software.

3.2. Chromium Browser

Chromium Browser Version 89.0.4389.90 was installed on the Ubuntu OS because it is a standard open-source browser that includes developer tools. Chromium can be directly launched from within Burp Suite Community edition with Burp interception pre-configured. When testing exploits for control purposes (non-isolated/native), navigation was direct to Webgoat using the URL http://172.17.0.1:8080/WebGoat/ because Webgoat was configured to listen on a real IP address rather than the loopback interface and TCP port 8080. ViewFinderJS used the URL http://127.0.0.1:8002 because ViewFinderJS was configured to listen on the loopback interface and TCP port 8002.

3.3. OWASP Webgoat

OWASP Webgoat 8.1.10 was installed on the Ubuntu system to create an attack target for testing. Webgoat is a highly exploitable web application developed by the Open Web Application Security Project (OWASP) to teach web application security by enabling student users to hack a vulnerable website. Webgoat includes instructive tutorials that teach how to hack websites and challenges that allow students to execute actual web application attacks. The default configuration of Webgoat is to bind to localhost (127.0.0.1) to reduce exposure of the vulnerable site. However, Webgoat was instead configured to bind to the Ubuntu OS's routable IP address to overcome issues with ViewFinderJS's inability to reach the host OS loopback due to docker networking issues.

3.4. Burp Suite Community Edition v2021.3.1 Build 6584

Burp Suite Community Edition v2021.3.1 Build 6584 was installed on the Ubuntu OS to intercept chromium web traffic for attack and recording purposes. Burp Suite is a pen-testing tool with a free community edition. Burp Suite acts as an HTML proxy and allows a security professional to intercept (machine-in-the-middle) a web application session. Once traffic is intercepted, the security professional can review and modify

HTML traffic on the fly before forwarding it as part of the overall web session. If Burp Suite is in the path, it records a history of HTML traffic and WebSocket traffic it observes.

3.5. ViewFinderJS / Docker Version called Browsergapce:2.6

ViewFinderJS is a "clientless, remote browser isolation product (RBI), including secure document viewing (CDR), built-in HTML/JavaScript that runs right in your browser" (Stringfellow, 2020). The solution was created by Cris Stringfellow and licensed under AGPLv3. The testing lab included a version of ViewFinderJS installed on as a docker image called BrowserGapCE 2.6. ViewFinderJS was chosen for remote browser isolation testing because it is free for non-commercial use, easing the burden for other researchers in replicating the results or expanding on this research. It was the only open-source RBI solution found that could support this research. The tested version was downloaded from Docker Hub using instructions on the ViewFinderJS GitHub page. ViewFinderJS only supports the pixel-pushing method of RBI, and therefore, all experimentation for the research was done only with pixel-pushing technology.

4. Experimentation and Analysis

To evaluate the potential and efficacy of RBI technology to be used as a reverse proxy to protect web applications from attack, it first had to be decided which types of web application attacks should be tested. The "OWASP Top 10 addresses the most impactful application security risks currently facing organizations" (OWASP, 2021). It is known industry-wide as the gold standard for the identification of the most critical web application security risks. The research was limited to confidentiality and integrity of the web applications and their associated sessions. Availability is an essential consideration but has been purposely left out of the scope of this research because availability protections are highly specific to architecture and are not well suited to be tested in a general-purpose proof of concept lab environment. Based on a review of the OWASP Top 10, it became clear that there were four significant areas of web application security risks that were likely to be mitigated by RBI technology. Experiments were devised around each of the risks. These risks are:

1. Attacks that involve modification of HTML requests or responses
2. Attacks that include injecting code into HTML forms
3. Attacking browsers by way of a vulnerable web application (e.g., using a website vulnerability to store or reflect a malicious script to attack clients)
4. The risk of data leakage to an untrusted and unauthorized device

Testing of exploitation of various web application vulnerabilities on Webgoat webserver through both an isolated connection (RBI) and a direct non-isolated connection for control was completed with the lab. The following flows show how tests were executed both with and without isolation.

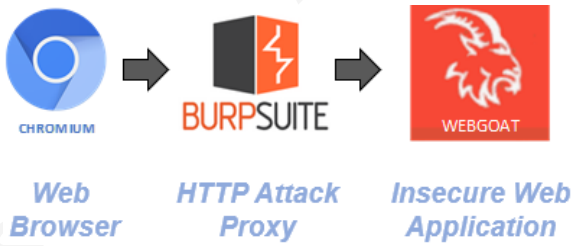


Figure 5 - Non-Isolated (Control) Testing Flow

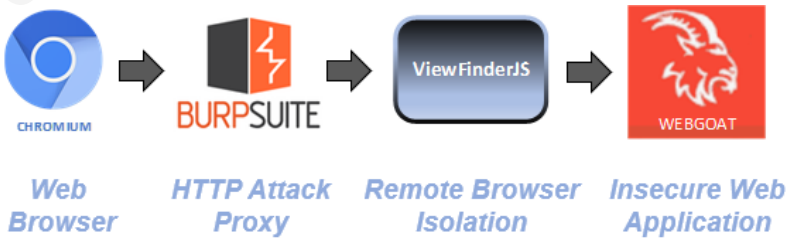


Figure 6 - Isolated (RBI) Testing Flow

4.1. Client-side HTML Tampering Experiment

Client-side HTML Tampering occurs when an attacker modifies HTML that is passed from the client to the server in a way that causes the web application to behave in ways unintended by the developer. This experiment was chosen for its ability to validate RBI's impact on any class of web application attack dependent on the ability to intercept and modify HTML requests and responses. Detailed notes and screenshots for this experiment can be found in Appendix A.

For this experiment, the Webgoat challenge that involves modifying an HTML POST to change the cost to purchase a television in a retail web application was used. The challenge consists of intercepting the HTML POST and replacing the original television value of $2999.99 for a lower value, such as $0.01, that due to lack of server-side validation, would be accepted.

For this experiment's non-isolated control, the Chromium browser was launched with the Burp proxy configured to intercept traffic. The Chromium browser was then navigated to Webgoat the HTML tampering challenge. The challenge was initially validated by submitting the form with no modification to the HTML. The result was a statement from Webgoat that the attack did not succeed which acted as a control to prove that additional testing would be valid. The form was submitted again, but this time, the HTML POST was modified to change the value $2999.99 to $0.01. The result was a statement from Webgoat that the attack had succeeded. Had this been an actual website, a malicious attacker might have walked away with a highly discounted TV!

To test isolation, the experiment was performed a second time with the ViewFinderJS RBI solution in the path. Chromium browser was launched with the Burp proxy configured to intercept traffic. Chromium browser was then navigated to the ViewFinderJS RBI solution. ViewFinderJS was then used to navigate to the Webgoat client-side HTML tampering challenge. The form was tested with no modification, and the result was Webgoat stating the attack had failed. The HTML intercept capability of Burp was then enabled, and an attempt was made to submit the form where the HTML POST could be modified as it was in the control test. Rather than intercept the HTML associated with the challenge, Burp only recorded HTML events related to mouse movements and frame buffer events between the Chromium browser and the ViewFinderJS RBI solution. Burp had no visibility to the actual web application HTML since the real page was being rendered on ViewFinderJS.

This experiment demonstrates that RBI technology can protect from attacks that involve client-side HTML modification or interception and modification of HTML traffic between the client and server in general.

4.2. Injection Experiment

Injection occurs when an adversary uses a web application to inject code or commands into another process or the operating system to generate a response unintended by the developer. There are several different types of injection, including SQL database injection, LDAP injection, and OS injection. This experiment was chosen for its ability to validate RBI's impact on a class of web application attacks that may involve attacking a web application through form input rendered in the client browser. Form input that can be manipulated directly within the rendered webpage may not require the HTML interception and modification described in the prior experiment. Detailed notes and screenshots for this experiment can be found in Appendix B.

For this experiment, the Webgoat challenge that involves injecting a crafted SQL statement into a web form was chosen. The purpose of the challenge is to demonstrate that if form input isn't properly validated, an attacker may be able to manipulate the input so that the backend system responds in unexpected ways. In this particular case, the goal is to cause the backend SQL server to dump the `user_system_data` table, which includes usernames and passwords.

For this experiment's non-isolated control, the Chromium browser was launched with the Burp proxy configured to pass and record traffic. The browser was then navigated to the Webgoat SQL injection challenge. The challenge was initially validated by submitting a test message into the injectable field. The response from Webgoat was a statement that no results were returned which acted as a control to prove additional testing would be valid. The form was then submitted again, but this time, the injectable field was populated with a crafted SQL statement meant to cause the database to dump the `user_system_data` table. The attack proved successful in that the contents of the table were dumped into the browser window. To complete the test, the table contents were copied onto the clipboard and pasted into a local text editor independent of the browser. The Burp history was also evaluated, and the HTML request and response that showed the contents of the password table were located.

To test isolation, the experiment was performed a second time but with the ViewFinderJS RBI solution in the path. With RBI present, the same control test was

performed, resulting in a statement from Webgoat that no results were returned. The form was then submitted again with the crafted SQL attack statement. The injection succeeded, and the browser displayed the contents of the `user_system_data` table. However, unlike the control test, it was impossible to copy the dumped table data to the clipboard through native Chromium. It should be noted that the RBI browser did have a built-in feature to copy data to the clipboard but the feature was additive and could be disabled in the code. It was also impossible to find the HTML request and response packets with the structured table data present because the Burp proxy between the Chromium client and the RBI solution only captured RBI-related traffic for user input and pixel-pushing of the isolated page.

This experiment demonstrates that injection attacks within the browser that do not require HTML modification are still possible with RBI present. However, this experiment also proved that RBI offers several mitigations to reduce the efficacy of such an attack in that the data output isn't presented in textual form.

4.3. Cross-Site Scripting (XSS) Experiment

Cross-Site Scripting (XSS) occurs when a vulnerable web server is used to reflect a script to the server's users. The XSS attack uses a client's trust in a web application to prompt the client web browser to execute malicious code. This experiment was chosen because, unlike the previous attacks, which ultimately target compromise of the web application itself, XSS attacks only use the web application as a component in an attack against client browsers that use the web application. The simplest example of an XSS attack is called reflected XSS, and it involves crafting a hyperlink with malicious JavaScript embedded. When clicked, the hyperlink would navigate the client browser to a destination website that can be exploited to reflect embedded JavaScript via HTML back down to the client due to XSS vulnerability. Since web browsers are supposed to execute JavaScript, the results of this attack are that malicious JavaScript is run. One typical example of exploitation would be convincing a client browser to send session cookies stored in the browser to an attacker-controlled listener on the Internet. XSS attacks can also be stored on the server. An example of a stored XSS is an attack where an authenticated user embeds malicious JavaScript in a page that other users are expected to

read at some point, such as a forum post. For this experiment, a Webgoat reflected XSS demonstration challenged was used. Detailed notes and screenshots for this experiment can be found in Appendix C.

For the non-isolated control of this experiment, the Chromium browser was launched with Burp configured to record traffic. The browser was then navigated to the appropriate XSS challenge page within Webgoat. A crafted JavaScript that would exploit an XSS vulnerability in the challenge page was placed in a form input field, and the form was submitted. The local Chromium browser executed the JavaScript and printed a text comment box as per the script instructions. The response proved the attack was viable in a non-isolated control.

To test isolation, the experiment was repeated with the RBI isolation in the path. The Chromium browser was again launched, but this time was initially navigated to the ViewFinderJS RBI. The RBI solution was navigated to the XSS challenge within Webgoat. The same crafted JavaScript was pasted into the challenge form and submitted. Like in the control test, the malicious JavaScript executed. However, this time the JavaScript was executed in ViewFinderJS and not on the local Chromium browser. ViewFinderJS responded with a different message, and the element view of Chromium developer tools was used to validate the message had originated as a pixel push from ViewFinderJS.

This experiment proved that XSS attacks are still possible with RBI present. However, this experiment also demonstrated that RBI technology can offer significant mitigations to reduce the efficacy of such an attack by moving the location where the script executes from the real browser to the RBI proxy. By executing the script in the RBI proxy, the true client is protected from running malicious code. Additionally, because any malicious code that does run within RBI is within an organizationally controlled environment, the surrounding architecture can be leveraged to further mitigate risks. Specific examples of architectural considerations to reduce risk will be discussed in the findings and discussion section below.

4.4. Data Leakage Experiment

Data leakage occurs when data is transferred to a device or user that is unauthorized to access the data. A common concern of browser-based zero-client remote access from an untrusted device is that the untrusted device may not be compliant to store and process data that a web application may make available for download. Take, for example, a webmail application like Outlook Web Access (OWA). Organizational users at third-party locations and without access to secure devices may have a significant mission-based justification to communicate through email. However, if access to OWA is granted to untrusted devices, it is possible that sensitive attachments within email messages could be downloaded to the untrusted device. This experiment was chosen to validate RBI's ability to protect from data leakage through downloading files. Webgoat did not have a file downloading exercise with a common file that would be at risk of data leakage like a PDF, word document, or CAD drawing. Instead, this test attempted to download an image displayed on a page within Webgoat. Detailed notes and screenshots for this experiment can be found in Appendix D.

For this experiment's non-isolated control, the Chromium browser was launched with the Burp proxy configured to intercept traffic. The Chromium browser was then navigated to a cross-site request forgeries page in Webgoat that included a JPG image of a cat. Using the right-click Chromium browser menu, the image was downloaded into the downloads folder in its original form. This control proved that through a standard browser, data leakage by way of direct file download was possible. Had this file contained sensitive metadata like geographical coordinates, the machine that downloaded the file would now have a copy of that metadata.

To test isolation, the experiment was performed again within the ViewFinderJS solution. After launching Chromium with Burp proxy enabled, the browser was navigated to ViewFinderJS. ViewFinderJS was then navigated to the same cross-site request forgeries page in Webgoat that included a JPG image of a cat. Attempting to right-click the cat image and download it resulted in a new and different menu generated by ViewFinderJS. The presented menu offered no way to download the original image. The screenshot option in the menu was tested, but that did not result in downloading the

original file but instead a screenshot of the full page that contained the rendered image. Since the original image could not be downloaded, at least metadata contained within it was protected by data leakage.

This experiment demonstrates that RBI could be used to prevent untrusted clients from causing data leakage events by downloading original data from protected web applications to unauthorized devices.

5. Findings and Discussion

Experimentation has shown that RBI provides significant mitigation to risks created by allowing untrusted client access to such applications. The table below summarizes the findings, followed by more detailed discussions of the findings and their ramifications.

Attack Method	Result	Detail
Client-side HTML Tampering	**Fully Mitigated**	*The attacker was unable to tamper with web application HTML traffic on the client-side while the RBI solution was in use.*
Injection Attack	**Partially Mitigated**	*Form-based injection was confirmed possible with RBI. However, impacts were mitigated by the lack of access to textual data either in the browser or in intercepted HTML.*
Cross-Site Scripting	**Partially Mitigated**	*XSS attack was successful at executing code within the RBI rendering engine. However, the impact of such an attack is mitigated by the fact that XSS would not execute in the client browser, which is the attack's true target. Moreover, risks to RBI by XSS can be mitigated via architectural means.*
Data Leakage Protection	**Fully Mitigated**	*RBI prevents data leakage through downloads. Functionality can be preserved for users through remote rendering.*

Table 1 - Experimentation Findings

The client-side HTML tampering experiment helped identify several significant findings. This experiment confirmed that ViewFinderJS was only pushing pixels and user interface movements back and forth between itself and the actual client browser. The experiment validated that no attack against a web server that was dependent on modification of web application HTML would be possible with a pixel-pushing RBI solution in the path. This is an enormous finding because it means the use of RBI could protect a vulnerable web application from an entire class of web application attacks being

launched from an insecure client through a man-in-the-browser attack or through an insecure network through a machine-in-the-middle attack. Additionally, this finding proves that RBI would eliminate the risk of an internal application unintentionally releasing information otherwise hidden from view in underlying HTML.

In the SQL injection experiment, the use of RBI did not outright block form-based injection. With RBI present, it was still possible to perform SQL injection through form-field entry to dump sensitive table contents. However, RBI did appear to offer several mitigations that would lower the risk and impact of a form-based injection attack against a web server. When SQL injection was tested with RBI, the dumped table data was not presented in textual format either within Chromium or Burp Suite. The dumped table data was only displayed visually as a rendered image in Chromium. Suppose an adversary was dumping the contents of a large table with hundreds to possibly tens of thousands of entries. In that case, the lack of textual output could pose a serious challenge to collecting and using the data. The adversary would need to find a way to collect the data either through manual data entry or optical character recognition (OCR) technology. While not an insurmountable challenge, this could slow down and frustrate an otherwise trivial attack.

Based on the previous observations, it can be surmised that most types of automated multi-step attack that depends on textual data would almost certainly fail. Consider an attack application that automates phishing using a typical enterprise webmail application like Microsoft Outlook Web Access (OWA) with password-only authentication. It would be possible for an adversary to use the HTML interface of OWA to write a scraper that, once authenticated as a user, automatically pulled the user's recent contacts. The tool could then send the compromised user's contacts a phishing email requesting that they provide their password to an attacker's web listener for some important reason. Suppose a user that received the phish provided their username and password. In that case, the same OWA scraper could then automatically harvest the new users contacts and start the process all over again. Since any type of textual-based scraping would be thwarted by pixel-pushing RBI technology, an attack like this would fail.

Craig Meyer, camsecure@fastmail.com

It is possible to develop a simple client-side JavaScript to sanitize the form input and block entry of characters such as dashes and quotes that make injection possible. Typically, it would not be effective to perform this mitigation on the client-side because an adversary could just remove or disarm the JavaScript through a tool like Burp Suite. However, with RBI present, disarming the JavaScript client-side would not be possible because the JavaScript would never make it to the actual client browser. Therefore, RBI could make an otherwise ill-advised solution a practical option to quickly fix an exposure.

The use of RBI did not block cross-site scripting, but the experiment proved that using RBI with a proper architecture could result in partial mitigation. This experiment showed that in an XSS attack, a malicious script would execute within the isolated browser rather than the native client browser. RBI browser execution of a malicious script results in three significant mitigations to the attack. First, since the execution occurs within the isolated browser, no information stored in the browser of the actual client would ever be at risk of collection. Similarly, no vulnerability within the actual client browser would ever be in danger of being exploited. Second, the RBI technology creates a direct mitigation against an XSS attack used for session hijacking. A common goal for a cross-site scripting attack is to collect a browser session token such as a cookie. The adversary can then take over the session and impersonate the user to hijack their authenticated session. Even if the XSS attack successfully delivers a session token to an adversary, such a token would be useless if the web application wasn't exposed directly to the Internet because it is placed behind RBI. Blocking exposure of a web application to the Internet would be a typical configuration in an architecture where RBI was used to isolate a web application. Finally, any XSS attack that involves sending session data to an adversary, such as a password entered into a form, can be mitigated through network packet filtering deployed by the organization in control of the RBI solution. Simply placing a firewall between the RBI solution and the Internet and configuring the firewall not to allow outbound sessions to originate from the RBI solution would block the XSS script from sending data.

Craig Meyer, camsecure@fastmail.com

The use of RBI could also be an effective way to mitigate against data leakage to unauthorized clients. The data leakage experiment suggests that RBI can prevent a client from downloading a file otherwise made available through a web application. Finally, since pixel-pushing RBI disables native clipboard copy through the client browser, data leakage via the clipboard can be partially mitigated. Again, the use of OCR may overcome this protection.

6. Recommendations and Implications

The research in this paper has demonstrated that Remote Browser Isolation technology can provide significant mitigation to the risks created by allowing untrusted browser clients to access organizational web applications. Organizations that must present web applications to external users on untrusted devices should consider the inclusion of RBI technology into their overall architecture. This technology offers an option to reduce the risk of a web application either being compromised or leaking data without blocking the site and outright preventing the use of the web application.

6.1. Recommendations for Practice

An ideal solution for any large organization that needs to present many applications to untrusted remote clients would be to build a secure web application presentation architecture. Such an architecture could start with an IDentity Provider (IDP) authentication landing page that first authenticated the user and posture assessed the client. Once the user's identity and the device's posture have been confirmed, a redirect could be configured to a web menu system that displayed the user's entitlements. Based on posture assessment, user access to web applications could be configured to be blocked, allowed with isolation, or allowed natively.

The following diagram depicts an entitlement system that could be built around RBI. In the entitlement system, an unsafe posture would be a device that could not be posture assessed or a device that failed a security check during the assessment process. A normal posture may be a device that has been posture assessed and meets a number of standard checks such as supported operating system, antivirus installed, antivirus is up to date, and running current patches, but is not fully controlled by the organization. An

enhanced posture would be the detection of an indicator that the parent organization secures the device. An example check that could be done is the identification of an organizationally issued X.509 device certificate. Based on these postures and categorization of whether each application represents a high, medium, or low risk, the entitlement system can choose between blocking access, providing isolated access, or allowing native access to the application.

Figure 7 - Sample RBI Entitlement Flow

Organizations considering building a web application presentation architecture that includes RBI should look for RBI features critical in such an architecture. The following list of features should be considered during the requirements gathering phase of architecture development.

1. RBI solution can be limited to authenticated users
2. RBI solution can support SAML both for local authentication and SSO to isolated web applications
3. RBI solution can support redirects to bring in isolated traffic from a menu system
4. RBI solution has a secure document rendering solution that supports common formats in use by organizations (e.g., TXT, PDF, ODT, ODS, DOC/DOCX, XLS/XLSX, PPT/PPTX, VSD/VSDX)
5. RBI solution that supports DOM tree mirroring must allow for granular control to push higher risk sites to pixel-pushing
6. RBI solution should have granular controls for configuring data leakage features such as enabling and disabling direct download and enabling and disabling copy-paste features
7. RBI solution for disabling copy-paste features should be implemented server-side and not in client-side JavaScript

6.2. Opportunities for Future Research

This research focused on proving a potential security benefit in leveraging RBI technology as a reverse proxy to protect organizational web applications from untrusted clients. However, the design and testing of specific architecture for deployment that includes an entitlement system and addresses all of the engineering challenges around redirection and identity management would be beneficial for organizations considering RBI for zero-client and zero-trust applications.

As described further in the paper, there is an alternative RBI implementation to pixel-pushing called DOM tree mirroring that would likely create some usability and security tradeoffs if used as a secure reverse proxy. Since a DOM tree mirror solution was not available as a free and open-source product, the technology was not tested for this research paper. Investigating how the use of DOM tree mirroring would impact the protections provided by RBI could help inform future architectures based on this concept.

Finally, it is recommended that vendors who specialize in web application security or remote access products consider including RBI technologies as reverse proxy solutions directly into their product ecosystems. Vendors that have already created an

RBI solution for client protection should consider enhancing their products with features targeted at the zero-client and zero-trust remote access market.

7. Conclusion

This research effort is intended to prove that Remote Browser Isolation (RBI) technology can provide a practical use to secure web applications from untrusted clients for zero-client and zero-trust remote access. The experiments performed to support this paper demonstrated that pixel-pushing RBI technology could offer meaningful protections to web applications instead of just clients where this technology is targeted. Further research would be needed to determine if DOM tree mirroring-based RBI would be similarly effective.

Craig Meyer, camsecure@fastmail.com

References

C. Stringfellow, (2020), *i5ik ViewFinderJS* [Source Code] https://github.com/i5ik/ViewFinderJS

Hiwarale. U. (2019, August). *How the browser renders a web page? — DOM, CSSOM, and Rendering.* Retrieved Medium.com: https://medium.com/jspoint/how-the-browser-renders-a-web-page-dom-cssom-and-rendering-df10531c9969

MacDonald. N. (2018). *Innovation Insight for Remote Browser Isolation – ID G00350577* Retrieved from Gartner, Inc. database

OWASP. (2021). *OWASP Top 10 – 2017 – The Ten Most Critical Web Application Security Risks*. Retrieved from OWASP Top 10 Project: https://owasp.org/www-pdf-archive/OWASP_Top_10-2017_%28en%29.pdf.pdf

Remington. D. (2020, January). *Cloudflare + Remote Browser Isolation.* Retrieved from The Cloudflare Blog: https://blog.cloudflare.com/cloudflare-and-remote-browser-isolation/

S. Rose et al., (2020, August). Zero Trust Architecture, NIST Special Publication 800-207, US Department of Commerce, National Institute of Standards and Technology, Gaithersburg, MD.

Verizon Enterprise. (2020). *2020 Data Breach Investigations Report.* Retrieved from Verizon.com: https://enterprise.verizon.com/resources/reports/2020-data-breach-investigations-report.pdf

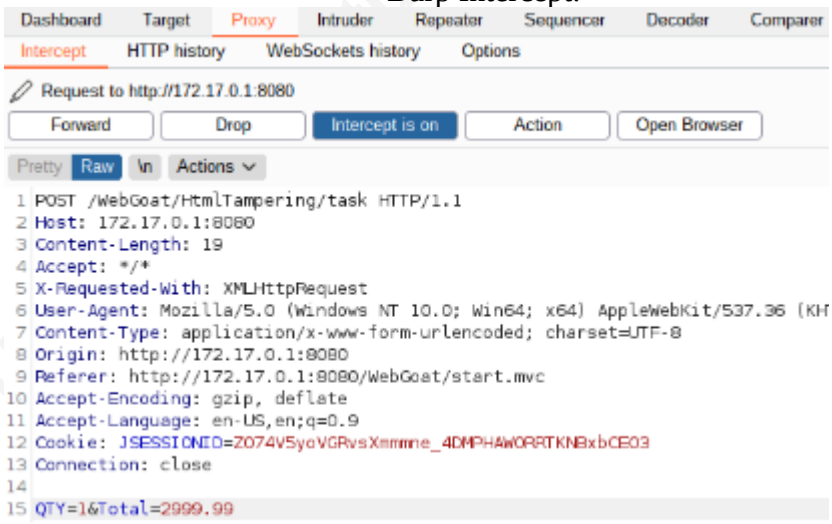
Craig Meyer, camsecure@fastmail.com

Appendix A – Client-Side HTML Tampering Experiment Notes

Client-Side HTML Tampering Experiment – Non-Isolated Control

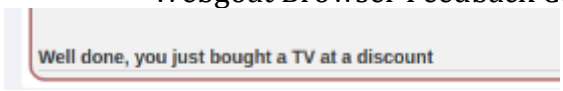
1. **Action:** Launch Burp Chromium browser
2. **Action:** Navigate to Webgoat and then navigate to Client side > HTML tampering > 2
3. **Action:** Click 'Checkout' to submit form and attempt to purchase TV at regular price
4. **Response:** Webgoat responds "This is too expensive... You need to buy at a cheaper cost!"
5. **Action:** Turn 'intercept on' in Burp
6. **Action:** Click 'Checkout' again to submit form
7. **Action:** Click forward on packets in Burp until packet is seen with 'QTY=1&Total=2999.99'. See screenshot:

Burp Intercept:



8. **Action:** Modify request to buy the TV at one cent by changing 'QTY=1&Total=2999.99' to 'QTY=1&Total=0.01' and then forward
9. **Response:** Webgoat responds "Well done, you just bought a TV at a discount" and response is also seen in Burp raw HTML response. See screenshots:

Webgoat Browser Feedback Confirming Attack Success:



Burp Modified Request and Response Proving Successful Attack:

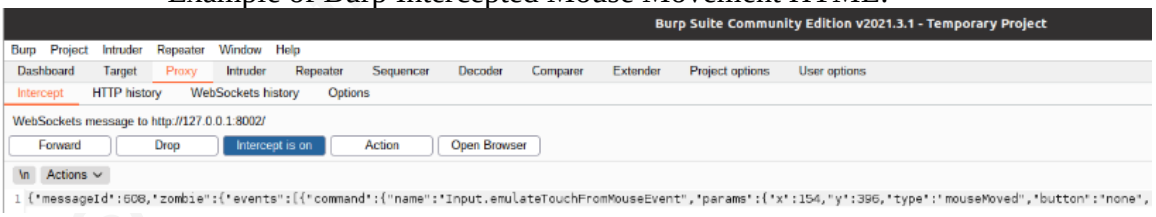
Craig Meyer, camsecure@fastmail.com

10. **Observation:** Test resulted in successful HTML tampering which resulted in a change in server-side action. Attack was executed through Burp interception of HTML and response could be seen in raw HTML being transported between client browser and Webgoat web application.

Client-Side HTML Tampering Experiment – Isolated with RBI

1. **Action:** Launch Burp Chromium browser
2. **Action:** Navigate to ViewFinderJS RBI
3. **Action:** Navigate to Webgoat and then navigate to Client side > HTML tampering > 2
4. **Action:** Click 'Checkout' to submit form and attempt to purchase TV at regular price
5. **Response:** Webgoat responds "This is too expensive... You need to buy at a cheaper cost!" in ViewFinderJS
6. **Action:** Turn on intercept in Burp Community Edition
7. **Observation:** Burp requires forwarding of many events covering input activities such as mouse movements and framebuffer events. See screenshot:

Example of Burp Intercepted Mouse Movement HTML:



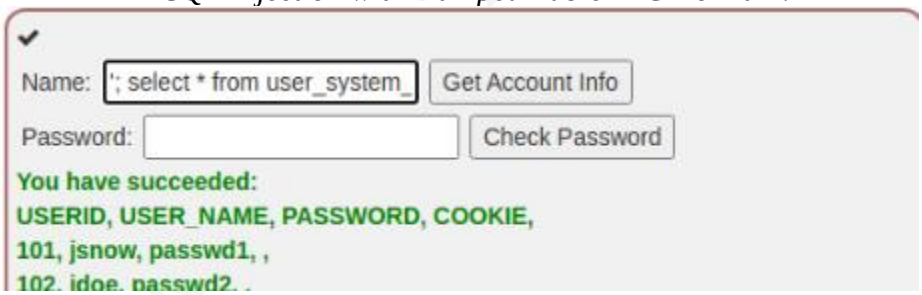
8. **Observation:** It is not possible to intercept any underlying HTML between the server and the ViewFinderJS client as Burp is only seeing WebSocket activity between the true client and ViewFinderJS; the attack is effectively stalled here. The act of intercepting the traffic makes the session basically unusable because the packet flow is just to fast for an attacker to continue to forward packets. In order to do any meaningful interception and modification at all during an RBI pixel push session the attacker would need to very carefully filter packets and act quickly. It may even require a pre-planned and scripted effort.

Appendix B - SQL Injection Experiment Notes

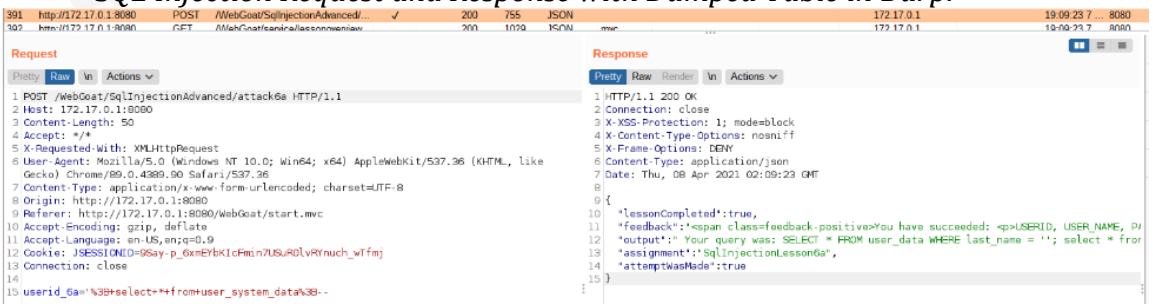
SQL Injection Experiment – Non-Isolated Control

1. **Action:** Launch Burp chromium browser
2. **Action:** Navigate to Webgoat and then navigate to (A1) Injection > SQL Injection Advanced > 3
3. **Action:** Type “test” into ‘Name:’ field and click ‘Get Account Info’
4. **Response:** Browser replies: “No results matched. Try Again.”
5. **Action:** Type “’; select * from user_system_data;--" into ‘Name:’ field and click ‘Get Account Info’
6. **Response:** Browser response shows a dump of a password table on screen and in HTML via Burp. See screenshots:

SQL Injection with Dumped Table in Chromium:



SQL Injection Request and Response with Dumped Table in Burp:



7. **Observation:** Test resulted in successful SQL injection printed to screen and captured via Burp proxy**.** Test could have been completed with proxy visibility and control alone. It was possible to natively copy the text to the clipboard from either the browser (right-click context menu) or through the Burp recorded response.

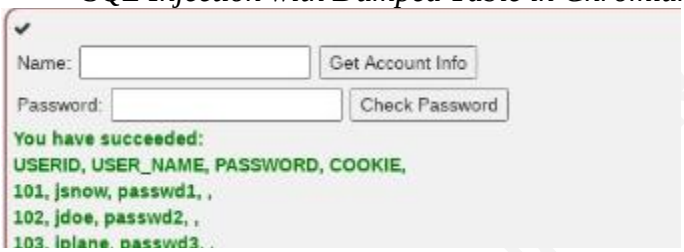
SQL Injection Experiment – Isolated with RBI

1. **Action:** Lunch Burp chromium browser
2. **Action:** Navigate to ViewFinderJS RBI
3. **Action:** Navigate to Webgoat and then navigate to (A1) Injection > SQL Injection Advanced > 3 within ViewFinderJS tab
4. **Action:** Type "test" into 'Name:' field and click 'Get Account Info'
5. **Response:** Browser replies: “No results matched. Try Again.”

Craig Meyer, camsecure@fastmail.com

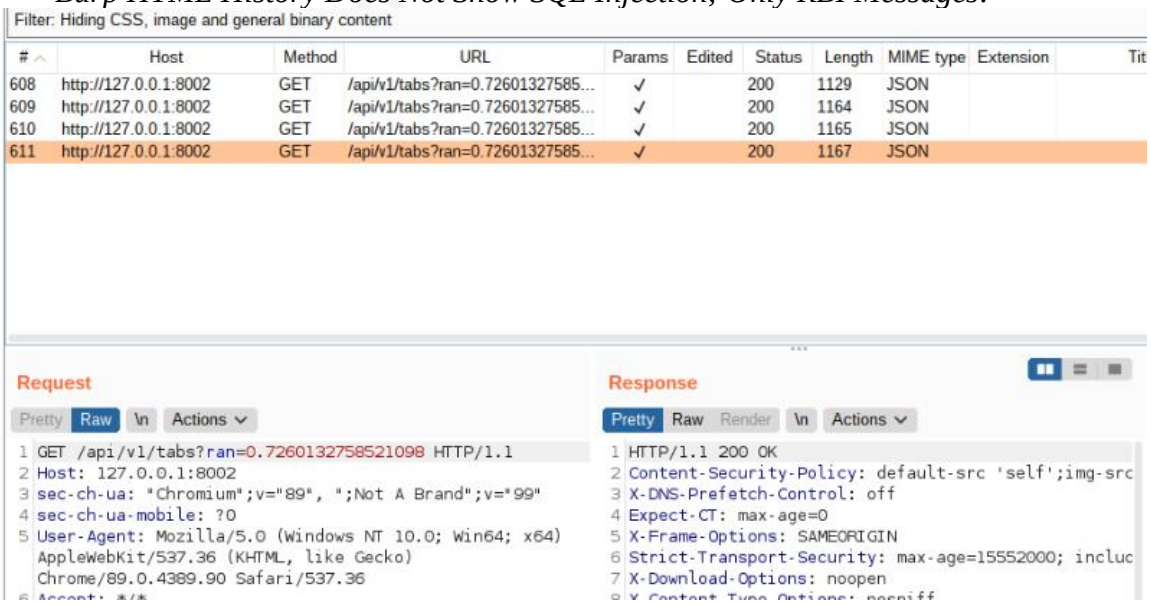
6. **Action:** Type '"; select * from user_system_data;--" into 'Name:' field and click 'Get Account Info'
7. **Response:** Browser response shows a dump of a password table on screen (in ViewFinderJS) but no corresponding HTML via Burp. See screenshot:

SQL Injection with Dumped Table in Chromium through RBI:

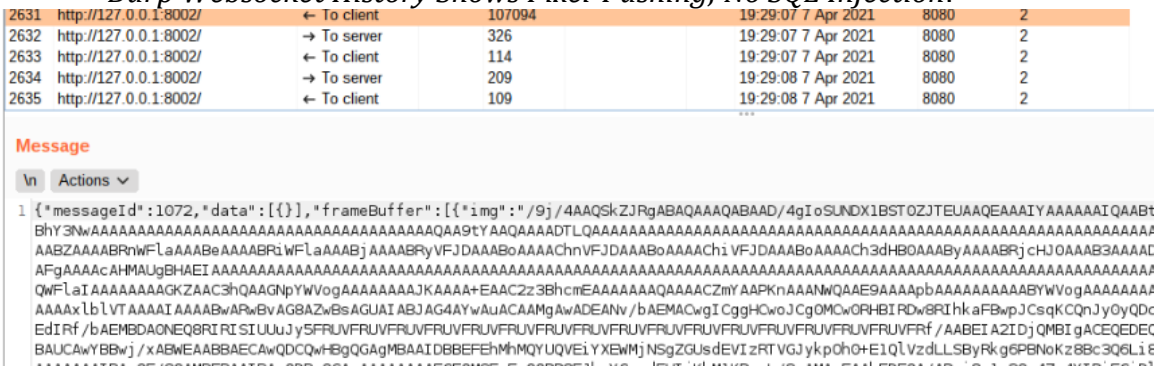


8. **Observation:** Burp does not capture attack; it only captures traffic related to RBI pixel-pushing (web sockets and mouse movements). See screenshots:

Burp HTML History Does Not Show SQL Injection; Only RBI Messages:



Burp Websocket History Shows Pixel-Pushing; No SQL Injection:



9. **Observation**: No POST method or response was seen at all within Burp; all typical HTTP based aspects of the injection attack were hidden from the Burp

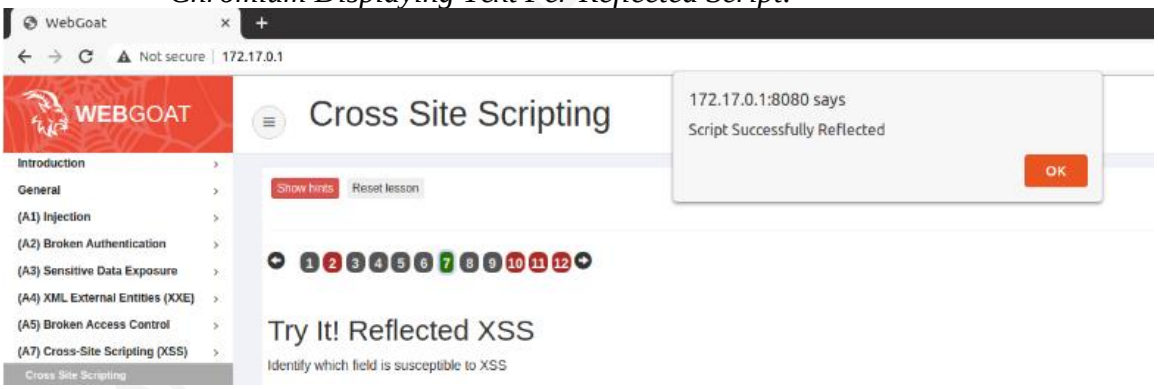
intercepted chromium client by the RBI solution. Still possible to inject and return data but only through a browser WebSocket which would be very hard to replicate from a secondary machine or through a man-in-the-browser attack. Through the browser the text was not interactive on the screen. In order to copy text to the clipboard the ViewFinderJS RBI provided a special context. However, since this menu is a feature built in the RBI it would be possible to disable it. Without network interception of HTML or the ability of the browser to capture data to the clipboard a large quantity of data would be very hard to capture and would require either manual transcription or Optical Character Recognition (OCR) which would frustrate an adversary's efforts.

Appendix C – Cross-Site Scripting (XSS) Experiment Notes

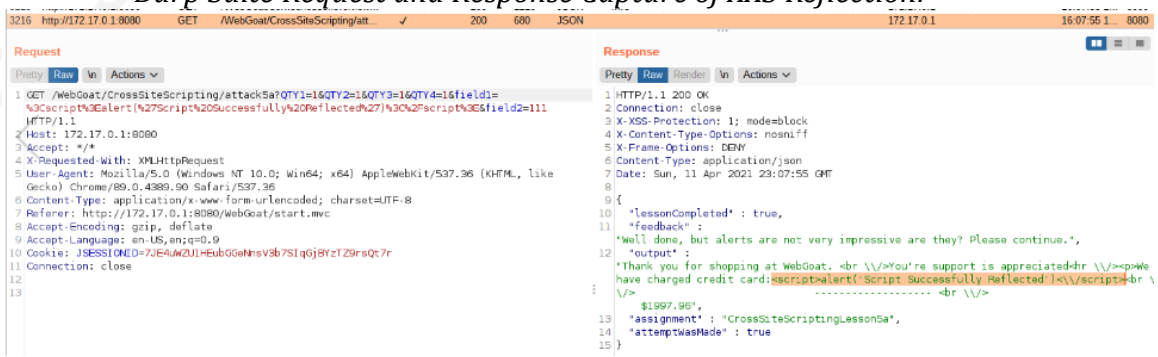
Cross-Site Scripting (XSS) Experiment – Non-Isolated Control

1. **Action:** Launch Burp chromium browser
2. **Action:** Navigate to Webgoat and then navigate to (A7) Cross-Site Scripting (XSS) > Cross-Site Scripting > 2
3. **Action:** Type "<script>alert('Script Successfully Reflected')</script>" into 'Enter your credit card number:' field and click 'Purchase' to submit the form
4. **Response:** Local Chromium displays "172.,17.0.1:8080 says Script Successfully Reflected". See screenshots:

Chromium Displaying Text Per Reflected Script:



Burp Suite Request and Response Capture of XXS Reflection:



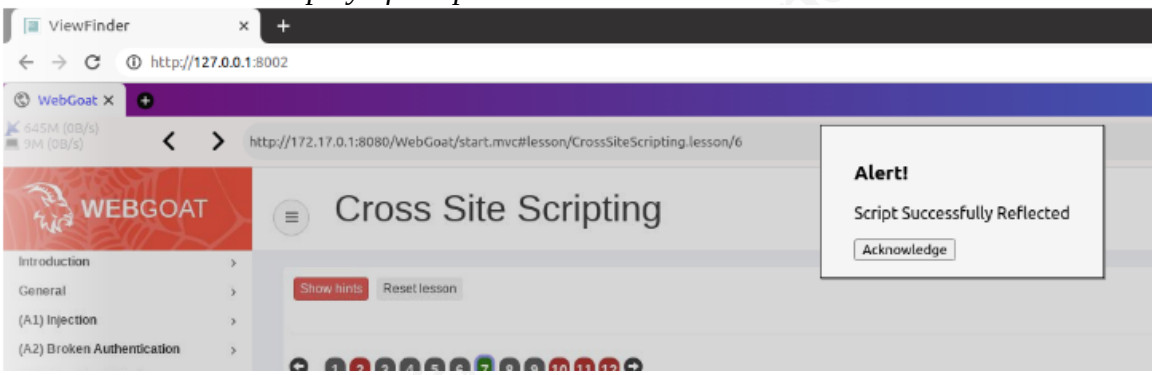
5. **Observation:** Test resulted in successful XSS reflection where JavaScript was executed in the local Chromium browser. HTML of request and response that included reflection can be captured and viewed in Burp.

Cross-Site Scripting (XSS) Experiment – Isolated with RBI

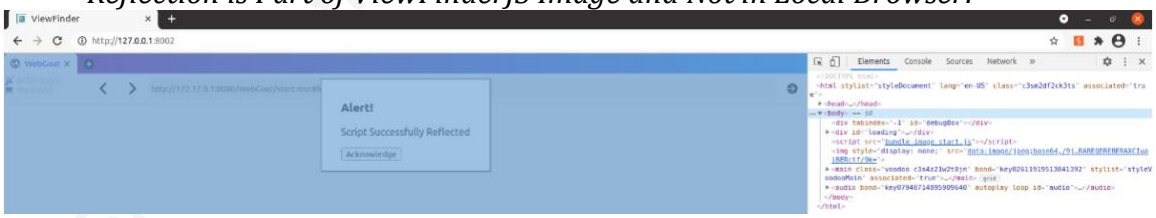
1. **Action:** Launch Burp chromium browser
2. **Action:** Navigate to ViewFinderJS RBI
3. **Action:** Navigate to Webgoat and then navigate to (A7) Cross-Site Scripting (XSS) > Cross-Site Scripting > 2
4. **Action:** Type "<script>alert('Script Successfully Reflected')</script>" into 'Enter your credit card number:' field and click 'Purchase' to submit the form

5. **Response:** Remote ViewFinderJS rendition displays "Alert! Script Successfully Reflected" but in the RBI environment and not local Chromium browser. See screenshots:

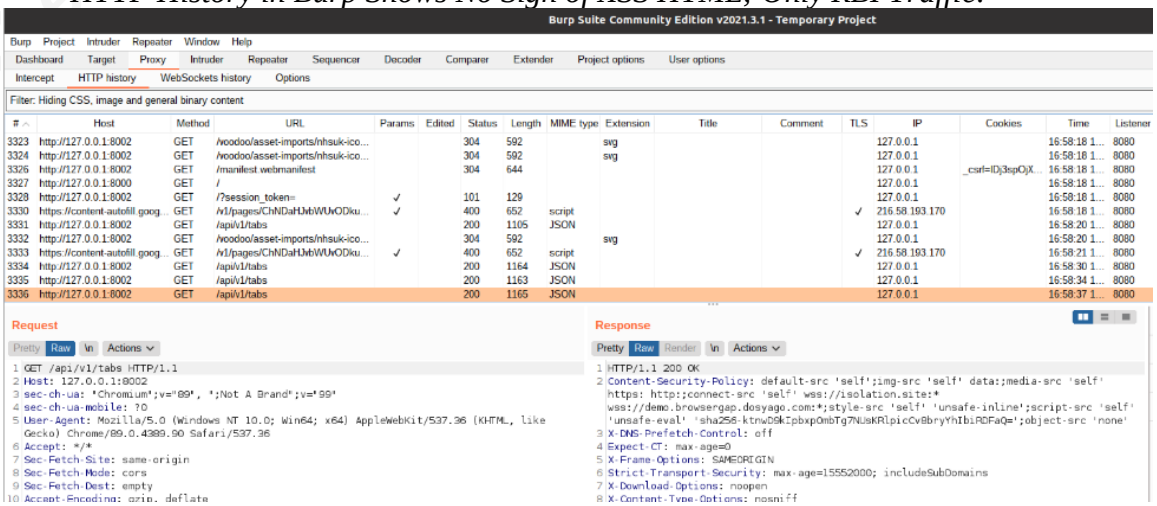
Browser Display of Script Alert Executed in ViewFinderJS:



Chromium Developer Tools Elements View of DOM Object; Proves XSS Reflection is Part of ViewFinderJS Image and Not in Local Browser:



HTTP History in Burp Shows No Sign of XSS HTML; Only RBI Traffic:



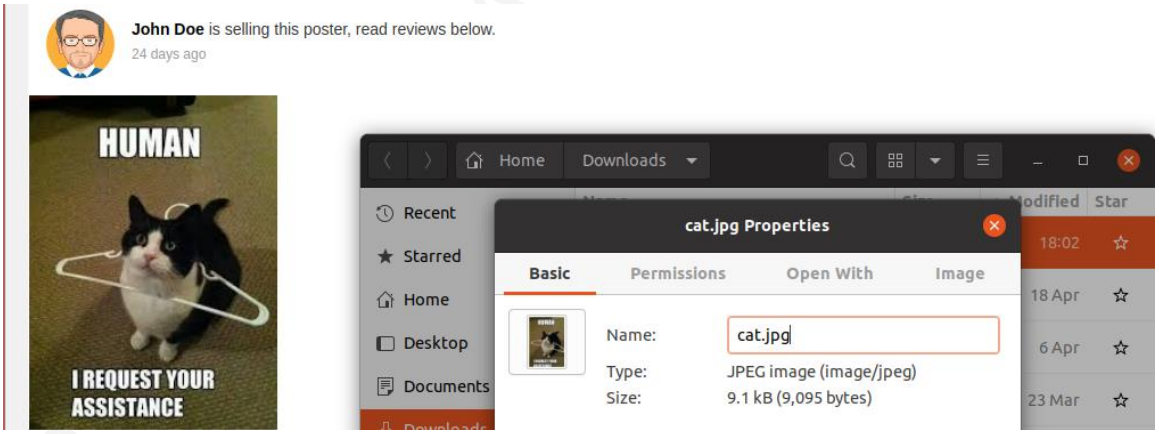
6. **Observation:** Test resulted in successful XSS reflection but JavaScript was executed but on the ViewFinderJS rendering engine rather than the local chromium browser. The dangers of XSS rendering on the true client are numerous including a local browser vulnerability, local stored data, and the lack of assurances that outbound connections to an attacker-controlled listener would be blocked. All of this can be well controlled in a managed RBI solution.

Appendix D – Data Leakage Experiment Notes

Data Leakage Experiment – Non-Isolated Control

1. **Action:** Launch Burp chromium browser
2. **Action:** Navigate to Webgoat and then navigate to (A8:2013) Request Forgeries > Cross-Site Request Forgeries > 24
3. **Action:** Right-click on the poster of a cat (embedded JPG image) and select "save image as…"
4. **Response:** Local Chromium downloads the original embedded file named cat.jpg and places it in downloads directory. See screenshot:

Chromium downloaded and stored file in downloads folder:

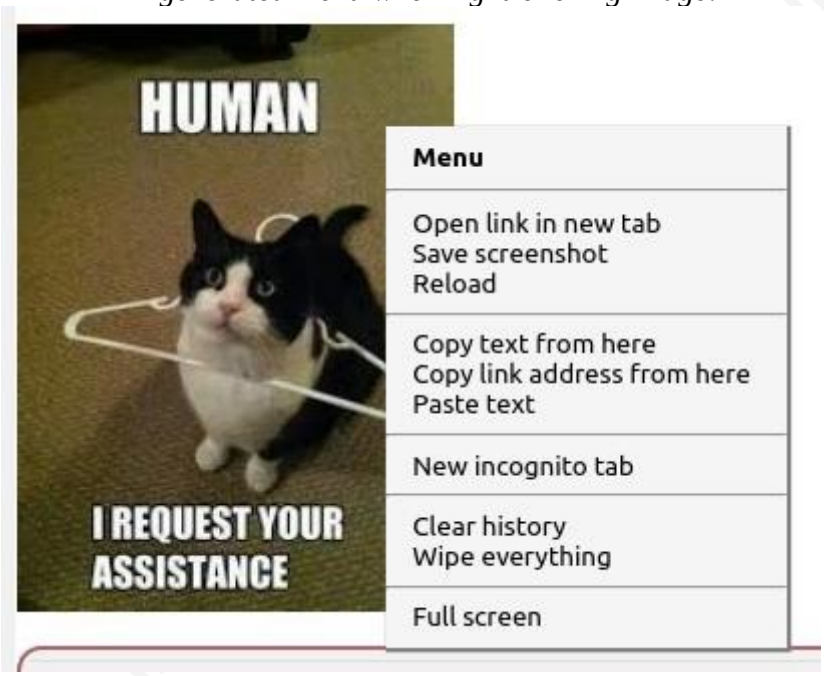


1. **Observation:** Test resulted in successful download of file. If this image file contained sensitive data such as geo-location data that identified the location this image was taken, data leakage would have occurred.

Data Leakage Experiment – Isolated with RBI

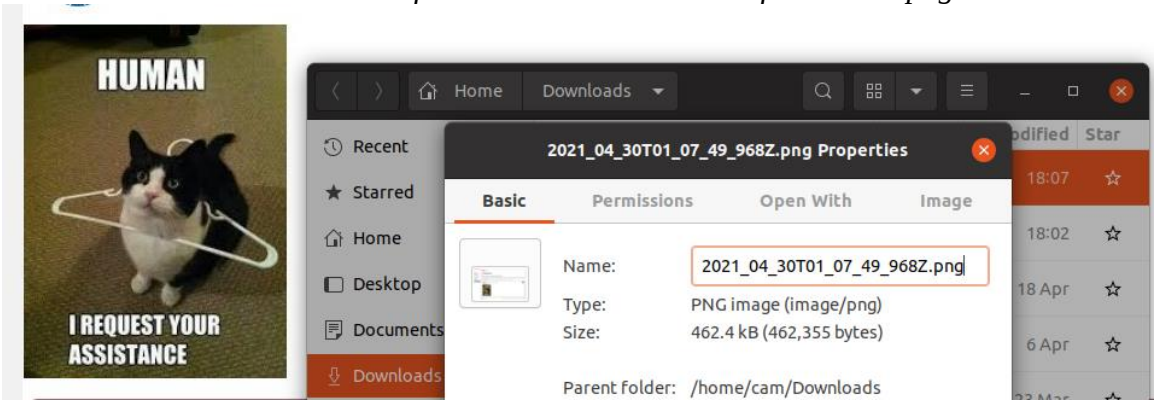
2. **Action:** Launch Burp chromium browser
3. **Action:** Navigate to ViewFinderJS RBI
4. **Action:** Navigate to Webgoat and then navigate to (A8:2013) Request Forgeries > Cross-Site Request Forgeries > 24
5. **Action:** Right-click on the poster of a cat (embedded JPG image)
6. **Response:** The RBI solution generates a menu on screen that has different options than the local browser when interacting with an image. No option to save the file exists on the menu. See Screenshot:

RBI generated menu when right-clicking image:



7. **Observation:** The RBI solution does not offer a means to directly download the image which appears to prevent data leakage such as image metadata. To further this experiment using the "Save screenshot" option was selected.
8. **Action:** Click "Save screenshot" menu option
9. **Response:** Local Chromium downloads an RBI generated screenshot of the entire visible page in the form of a PNG image and places it in downloads directory. See screenshot:

ViewFinderJS screenshot feature took a screenshot of the entire page:



10. **Observation:** The screenshot that was downloaded was simply a copy of the image rendered on the RBI browser and not the original image. Because the image cannot be downloaded directly at least some of the data included in the original file appears to be protected from leakage.