Puppetnets: Misusing Web Browsers as a Distributed Attack Infrastructure

Spiros Antonatos
FORTH-ICS, Greece
antonat@ics.forth.gr

Periklis Akritidis
Cambridge University, UK
Periklis.Akritidis@cl.cam.ac.uk

Vinh The Lam
University of California, San Diego, USA
vtlam@cs.ucsd.edu

Kirk Jon Khu
I2R, Singapore
kjkhu@i2r.a-star.edu.sg

Kostas G. Anagnostakis
I2R, Singapore
kostas@i2r.a-star.edu.sg

Most of the recent work on Web security focuses on preventing attacks that *directly* harm the browser's host machine and user. In this paper we attempt to quantify the threat of browsers being *indirectly* misused for attacking third parties. Specifically, we look at how the existing Web infrastructure (e.g., the languages, protocols, and security policies) can be exploited by malicious or subverted websites to remotely instruct browsers to orchestrate actions including denial of service attacks, worm propagation and reconnaissance scans. We show that attackers are able to create powerful botnet-like infrastructures that can cause significant damage. We explore the effectiveness of countermeasures including anomaly detection and more fine-grained browser security policies.

Categories and Subject Descriptors: D.4.6 [**Operating Systems**]: Security and Protection—*Invasive software*

General Terms: Security, Measurement, Experimentation

Additional Key Words and Phrases: Web security, malicious software, distributed attacks

Part of this work was performed while P. Akritidis and S. Antonatos were visiting I^{2}R under an industrial attachment programme, and while V.T. Lam was a research engineer with I^{2}R.

1. INTRODUCTION

In the last few years researchers have observed two significant changes in malicious activity on the Internet [Schneier 2005; Williams and Heiser 2004; Symantec 2005]. The first is the shift from amateur proof-of-concept attacks to professional profit-driven criminal activity. The second is the increasing sophistication of the attacks. Although significant efforts are made towards addressing the underlying vulnerabilities, it is very likely that attackers will try to adapt to *any* security response, by discovering new ways of exploiting systems to their advantage [Nachenberg 1997]. In this arms race, it is important for security researchers to proactively explore and mitigate new threats before they materialize.

This paper discusses one such threat, for which we have coined the term *puppetnets*. Puppetnets rely on websites that *coerce* web browsers to (unknowingly) participate in malicious activities. Such activities include distributed denial-of-service, worm propagation and reconnaissance probing, and can be engineered to be carried out in stealth, without any observable impact on an otherwise innocent-looking website. Puppetnets exploit the high degree of flexibility granted to the mechanisms comprising the web architecture, such as HTML and JavaScript. In particular, these mechanisms impose few restrictions on how remote hosts are accessed. A website under the control of an attacker can thereby transform a collection of web browsers into an *impromptu* distributed system that is effectively controlled by the attacker. Puppetnets expose a deeper problem in the design of the web. The problem is that the security model is focused almost exclusively on protecting browsers and their host environment from malicious web servers, as well as servers from malicious browsers. As a result, the model ignores the potential of attacks against third parties.

Websites controlling puppetnets could be either legitimate sites that have been subverted by attackers, malicious "underground" websites that can lure unsuspected users by providing interesting services (such as free web storage, illegal downloads, etc.), or websites that openly invite users to participate in vigilante campaigns. We must note however that puppetnet attacks are different from previous vigilante campaigns against spam and phishing sites that we are aware of. For instance, the Lycos "Make Love Not Spam" campaign [TechWeb.com 2004] required users to install a screensaver in order to attack known spam sites. Although similar campaigns can be orchestrated using puppetnets, in puppetnets users may not be aware of their participation, or may be coerced to do so; the attack can be launched stealthily from an innocent-looking web page, without requiring any extra software to be installed, or any other kind of user action.

Puppetnets differ from botnets in three fundamental ways. First, puppetnets are not heavily dependent on the exploitation of specific implementation flaws, or on social engineering tactics that trick users into installing malicious software on their computer. They exploit *architectural* features that serve purposes such as enabling dynamic content, load distribution and cooperation between content providers. At the same time, they rely on the *amplification* of vulnerabilities that seem insignificant from the perspective of a single browser, but can cause significant damage when abused by a popular website. Thus, it seems harder to eliminate such a threat in similar terms to common implementation flaws, especially if this would require sacrificing functionality that is of great value to web designers. Additionally, even if we optimistically assume that major security problems such as code injection and traditional botnets are successfully countered, some puppetnet attacks will still be possible. Furthermore, the nature of the problem implies that the attack vector is pervasive: puppetnets can instruct *any* web browser to engage in malicious activities.

Second, the attacker does not have complete control over the actions of the participating nodes. Instead, actions have to be composed using the primitives offered from within the browser sandbox – hence the analogy to puppets. Although the flexibility of puppetnets seems limited when compared to botnets, we will show that they are surprisingly powerful.

Finally, participation in puppetnets is dynamic, making them a moving target, since users join and participate unknowingly while surfing the net. Thus, it seems easy for the attackers to maintain a reasonable population, without the burden of having to look for new victims. At the same time, it is harder for the defenders to track and filter out attacks, as puppets are likely to be relatively short-lived.

A fundamental property of puppetnet attacks, in contrast to most web attacks that directly harm the browser's host machine, is that they only *indirectly* misuse browsers to attack third parties. As such, users are less likely to be vigilant, less likely to notice the attacks, and have lesser incentive to address the problem. Similar problems arise at the server side: if puppetnet code is installed on a website, the site may continue to operate without any adverse consequences or signs of compromise (in contrast to defacement and other similar attacks), making it less likely that administrators will react in a timely fashion.

In this paper we experimentally assess the threat from puppetnets. We discuss the building blocks for engineering denial-of-service attacks, worm propagation and other puppetnet attacks, and attempt to quantify how puppetnets would perform. Finally, we examine various options for guarding against such attacks.

2. PUPPETNETS: DESIGN AND ANALYSIS

We attempt to map out the attackers' opportunity space for misusing Web browsers. In lieu of the necessary formal tools for analyzing potential vulnerabilities, neither the types of attacks nor their specific realizations are exhaustive enough to provide us with a solid worst-case scenario. Nevertheless, we have tried to enhance the attacks as much as possible, in an attempt to approximately determine in what ways and to what effect the attacker could capitalize on the underlying vulnerabilities. In the rest of this section, we explore in more detail a number of ways of using puppetnets, and attempt to quantify their effectiveness.

2.1 Distributed Denial of Service

The flexibility of Web architecture provides many ways for launching DoS attacks using puppetnets. The common component of the attack in all of its forms is an instruction that asks the remote browser to access some object from the victim. There are several ways of embedding such instructions in an otherwise legitimate webpage. The simplest way is to add an image reference, as commonly used in the vast majority of webpages. Other ways include opening up pop-up windows, creating new frames that load a remote object, and loading image objects through JavaScript. We are not aware of any browser that imposes restrictions on the location or type of the target referenced through these mechanisms.

We assume that the intention of the attacker is to maximize the effectiveness of the DDoS attack, at the lowest possible cost, and as stealthily as possible. An attack may have different objectives: maximize the amount of ingress traffic to the victim, the egress traffic from the victim, connection state, *etc*. Here we focus on raw bandwidth attacks in both

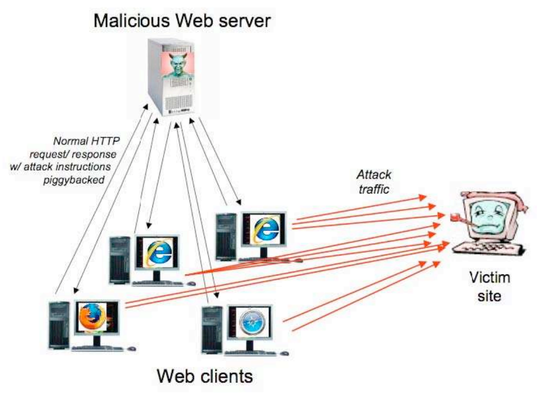

Malicious Web server
Normal HTTP request/ response w/ attack instructions piggybacked
Attack traffic
Victim site
Web clients

Fig. 1. **DDoS using puppetnets**

```
<SCRIPT>
pic= new Image(10,10);

function dosround() {
 var now = new Date();
 pic.src='http://target/xx?'+now.getTime();
 setTimeout ( "dosround()", 20 );
 return;
}
</SCRIPT>

<DIV id="rootDIV">
<IFRAME name='parent1' width="0%"
 src="originalpage.html"
 onLoad="dosround()">
</IFRAME></DIV>
```

Fig. 2. **Sample code for puppetnet DDoS attack**

directions, but emphasize on ingress traffic as it seems harder to defend against: the host has full control over egress traffic, but usually limited control over ingress traffic.

To create a large number of requests to the target site, the attacker can embed a sequence of image references in the malicious webpage. This can be done using either a sequence of `IMG SRC` instructions, or a JavaScript loop that instructs the browser to load objects from the target server. In the latter case, the attack seems to be much more efficient in terms of *attack gain*, e.g., the effort (in terms of bandwidth) that the attacker has to spend for generating a given amount of attack traffic. This assumes that the attacker either targets the same URL in all requests, or is able to construct valid target URLs through JavaScript without wasting space for every URL. To prevent client-side caching of requests, the attacker can also attach an invariant modifier string to the attack URL that is ignored by the server [1] but

[1] The URL specification [Berners-Lee et al. 1994] states that URLs have the form

considered by the client in the context of deciding whether the object is already cached.

Another constraint is that most browsers impose a limit on the number of simultaneous connections to the same server. For MSIE and Firefox the limit is two connections. However, we can circumvent this limit using aliases of the same server, such as using the DNS name instead of the IP address, stripping the "www" part from or adding a trailing dot to the host name, etc. Most browsers generally treat these as different servers. Servers that host more than one website (i.e., "virtual hosting") are especially vulnerable to this form of amplification.

To make the attack stealthy in terms of not getting noticed by the user, the attacker can employ hidden (*e.g.,* zero-size) frames to launch the attack-bearing page in the background. To maximize effectiveness, the requests should not be rendered within a frame and should not interfere with normal page loading. To achieve this, the attacker can employ the same technique used by Web designers for pre-loading images for future display on a webpage. The process of requesting target URLs can then be repeated through a loop or in an event-driven fashion. The loop is likely to be more expensive and may interfere with normal browser activity as it may not relinquish control frequently enough for the browser to be used for other purposes. The event-driven approach, using JavaScript timeouts, appears more attractive [2]. An example of the attack source code is shown in Figure 2.

2.1.1 *Analysis of DDoS attacks.* We explore the effectiveness of puppetnets as a DDoS infrastructure. The "firepower" of a DDoS attack will be equal to the number of users concurrently viewing the malicious page on their web browser (henceforth referred to as *site viewers*) multiplied by the amount of bandwidth each of these users can generate towards the target server. Considering that some web servers are visited by millions of users every day, the scale of the potential threat becomes evident. We consider the *size* of a puppetnet to be equal to the site viewers for the set of web servers controlled by the same attacker. Although it is tempting to use puppetnet size for a direct comparison to botnets, a threat analysis based on such a comparison alone may be misleading. Firstly, a bot is generally more powerful than a puppet, as it has full control over the host, in contrast to a puppet that is somewhat constrained within the browser sandbox. Secondly, a recent study [Cooke et al. 2005] observes a trend towards smaller botnets, suggesting that such botnets may be more attractive, as they are easier to manage and keep undetected, yet powerful enough for the attacker to pursue his objectives. Finally, an attacker may construct a hybrid, two-level system, with a small botnet consisting of a number of web servers, each controlling a number of puppets.

To estimate the firepower of puppetnets we could rely on direct measurements of site viewers for a large fraction of the websites in the Internet. Although this would be ideal in terms of accuracy, to the best of our knowledge there is no published study that provides such data. Furthermore, carrying out such a large-scale study seems like a daunting task. We therefore obtain a rough estimate using "second-hand" information from website reports and Web statistics organizations. There are several sources providing data on the number of daily or monthly visitors:

—Many sites use tools such as Webalizer [Barrett 2005] and WebTrends [Inc. 2006] to

`http://host:port/path?searchpart`. The `searchpart` is ignored by web servers such as Apache if included in a normal file URL.

[2]The referrer field can sometimes be scrubbed. See Section 3.

generate usage statistics in a standard format. This makes them easy to locate through search engines and to automatically post-process. We have obtained WebTrends reports for 249 sites and Webalizer reports for 738 sites, covering a one-month period in December 2005. Although these sites may not be entirely random, as the sampling may be influenced by the search engine, we found that most of them are non-commercial sites with little content and very few visits.

—Some Web audit companies such as ABC Electronic provide public databases for a fairly large number of their customers [ABC Electronic 2006]. These sites include well-known and relatively popular sites. We obtained 138 samples from this source.

—Alexa [Alexa Internet Inc. 2006] tracks the access patterns of several million users through a browser-side toolbar and provides, among other things, statistics on the top-500 most popular sites. Although Alexa statistics have been criticized as inaccurate because of the relatively small sample size [Anonymous 2004], this problem applies mostly to less popular sites and not the top-500. Very few of these sites are tracked by ABC Electronic.[3]

We have combined these numbers to get an estimate on the number of visitors per day for different websites. Relating the number of visitors per day to the number of site viewers is relatively straightforward. Recall that Little's law [Little 1961] states that if λ is the arrival rate of clients in a queuing system and T the time spent in the system, then the number of customers active in the system N is $N = \lambda T$.

To obtain the number of site viewers we also need, for each visit to a website, the time spent by users viewing pages on that site. None of the sources of website popularity statistics mentioned above provide such statistics. We therefore have to obtain a separate estimate of web browsing times, making the assumption that site popularity and browsing times are not correlated in a way that could significantly distort our rough estimates.

Note that although we base our analysis on estimates of "typical" website viewing patterns, the attacker may also employ methods for increasing viewing times, such as incentivizing users (*e.g.,* asking users to keep a pop-up window open for the download to proceed), slowing down the download of legitimate pages, and providing more interesting content in the case of a malicious website.

Web session time measurements based entirely on server log files may not accurately reflect the time spent by users viewing pages on a particular website. These measurements compute the session time as the difference between the last and first request to the website by a particular user, and often include a timeout threshold between requests to distinguish between different users. The remote server usually cannot tell whether a user is *actively* viewing a page or whether he has closed the browser window or moved to a different site. As we have informally observed, many users leave several browser instances open for long periods of time, we were concerned that web session measurements may not be reliable enough by themselves for the purposes of this study. We thus considered the following three data sources for our estimates:

—We obtain *real* browsing times through a small-scale experiment: we developed browser

[3]Alexa only provides relative measures of daily visitors as a fraction of users that have the Alexa toolbar installed, and not absolute numbers of daily visitors. To obtain the absolute number of daily visitors we compare the numbers from Alexa to those from ABC Electronic, for those sites that appear on both datasets. This gives us a (crude) estimate of the Internet population, which we then use to translate visit counts from relative to absolute.

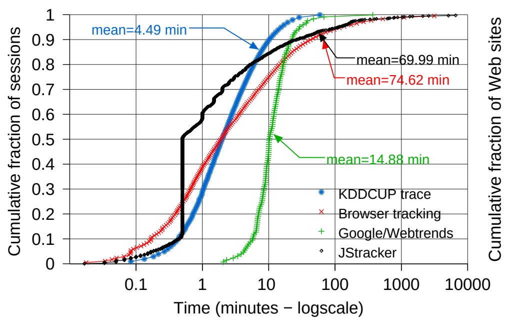


Fig. 3. **Website viewing times**

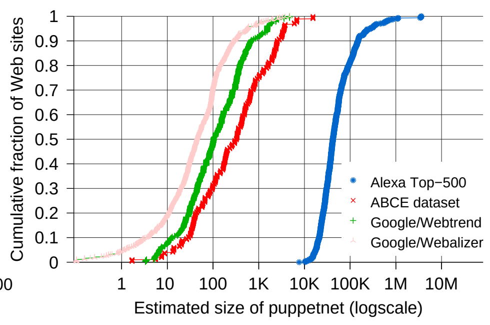


Fig. 4. **Estimated size of puppetnets**

extensions for both MSIE and Firefox that keep track of webpage viewing times and regularly post anonymized usage reports back to our server. The experiment involved roughly 20 users and resulted in a dataset of roughly 9,000 page viewing reports.

—We instrumented all pages on the server of our institution to include JavaScript code that makes a small request back to the server every 30 seconds. This allows us to infer how long the browser points to one of the instrumented pages. We obtained data on more than 3,000 sessions over a period of two months starting January 2006. These results are likely to be optimistic, as the instrumented website is not particularly deep or content-heavy.

—We analyzed the KDD Cup 2000 dataset [Kohavi et al. 2000] which contains clickstream and purchase data from a defunct commercial website. The use of cookies, the size of the dataset, and the commercial nature of the measured website suggest that the data are reasonably representative for many websites.

—We obtained, through a search engine, WebTrends reports on web session times from 249 sites, similar to the popularity measurements, which provide us with mean session time estimates per site.

The distributions of estimated session times, as well as the means of the distributions, are shown in Figure 3[4]. As suspected, the high-end tail of the distribution for the more reliable browser-tracking measurements is substantially larger than that for other measurement methods. This confirms our informal observation that users tend to leave browser windows open for long periods of time, and our concern that logfile-based session time measurements may underestimate viewing times. The JavaScript tracker numbers also appear to confirm this observation. As in the case of DDoS, we are interested in the mean number of active viewers. Our results show that because of the high-end tails, the mean time that users keep pages on their browser is around 74 minutes, 6-13 times more than the session time as predicted using logfiles[5].

From the statistics on daily visits and typical page viewing times we estimate the size of a puppetnet. The results for the four groups of website popularity measurements are

[4]We highlight the means as more informative in this case, as medians hide the high-end contributors to the aggregate page viewing time in a puppetnet.

[5]Note that the WebTrends distribution seems to have much lower variance and a much higher median than the other two sources. This is an artifact, as for WebTrends we only have access to a distribution of *means* for different sites, rather than the distribution of session times.

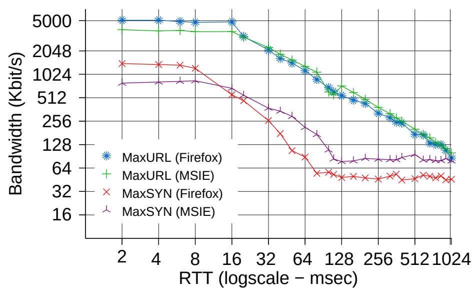


Fig. 5. **Ingress bandwidth consumed by one puppet vs. RTT between browser and server with the attack code based on JavaScript**

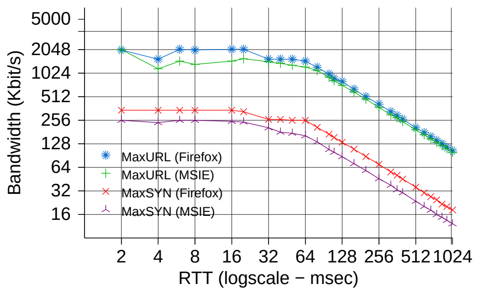


Fig. 6. **Ingress bandwidth consumed by one puppet vs. RTT between browser and server with the attack code based on static HTML code**

	Firefox	**MSIE**
maxSYN 2 aliases	83.97	106.30
maxSYN 3 aliases	137.26	173.28
maxURL 2 aliases	664.74	502.06
maxURL 3 aliases	1053.79	648.33

Table I. **Estimated bandwidth (Mbit/s) of ingress DDoS from 1000 puppets**

shown in Figure 4. The main observation here is that puppetnets appear to be comparable in size to botnets. Most top-500 sites appear highly attractive as targets for setting up puppetnets, with the top-100 sites able to form puppetnets controlling more than 100,000 browsers at any time. The sizes of the largest potential puppetnets (for the top-5 sites) seem comparable to the largest botnets seen [Keizer 2005], at 1-2M puppets. Although one could argue that top sites are more likely to be secure, the figures for sites other than the top-500 are also worrying: More than 20% of typical commercial sites can be used for puppetnets of 10,000 nodes, while 4-10% of randomly selected sites can be popular enough for hosting puppetnets of more than 1,000 nodes.

As discussed previously, however, the key question is not how big a puppetnet is but whether the firepower is sufficient enough for typical DDoS scenarios. To estimate the DDoS firepower of puppetnets we first need to determine how much traffic a browser can typically generate under the attacker's command.

We experimentally measure the bandwidth generated by puppetized browsers, focusing initially only on ingress bandwidth, since it is harder to control. Early experiments with servers and browsers in different locations (not presented here in the interest of space) show that the main factor affecting DoS strength is the RTT between client and server. We therefore focus on precisely quantifying DoS strength in a controlled lab setting, with different line speeds and network delays emulated using *dummynet* [Rizzo 1997], and an Apache webserver running on the victim host. We consider two types of attacks: a simple attack aiming to maximize SYN packets (maxSYN), and one aiming to maximize the ingress bandwidth consumed (maxURL). For the maxSYN attack, the sources of ten JavaScript image objects are set to be non-existent URLs repeatedly every 50 milliseconds. Every time the script writes into the image source URL variable, the browser stalls old connections are stalled and establishes new connections to fetch the newly-set image URL.

For the maxURL attack we load a page with several thousand requests for non-existent URLs of 2048 bytes each (as MSIE can handle URLs of up to 2048 characters). The link between puppet and server was set to 10 Mbit/s in all experiments.

In Figure 5, the ingress bandwidth of the server is plotted against the RTT between the puppet and the server, for the case of 3 aliases. The effectiveness of the attack decreases for high RTTs, as requests spend more time "in-flight" and the connection limit to the same server is capped by the browser. For the maxSYN experiment, a puppet can generate up to 300 Kbit/s to 2 Mbit/s when close to the server, while for high RTTs around 250 msec the puppet can generate only around 60 Kbit/s. For the maxURL attack, these numbers become 3-5 Mbit/s and 200-500 Kbit/s respectively. The results seem to differ for both browsers: MSIE generates more attack traffic than Firefox for maxSYN, while Firefox generates more traffic for maxURL. We have not been able to determine the exact cause of the difference, mostly due to the lack of source code for MSIE. The same figures apply for slower connections, with RTTs remaining the dominant factor determining puppet DoS performance.

We also considered the case where a puppet has JavaScript disabled. We redesigned the attack page so that it only includes HTML code without any dynamic object or JavaScript code. The results are summarized in Figure 6. We notice a 30% reduction in the effectiveness of DDoS attack in the absence of JavaScript for maxURL, while for maxSYN the attack is up to 50% less potent.

Using the measurements of Figure 5, the distribution of RTTs measured in [Smith et al. 2003] and the capacity distribution from [Saroiu et al. 2002], we estimate the firepower of a 1000-node puppetnet, for different aliasing factors, as shown in Table I. From these estimates we also see that around 1000 puppets are sufficient for consuming a full 155 Mbit/s link using SYN packets alone, and only around 150 puppets are needed for a maxURL attack on the same link. These estimates suggest that puppetnets can launch powerful DDoS attacks and should therefore be considered as a serious threat.

Considering the analysis above, we expect the following puppetnet scenarios to be more likely. An attacker controlling a popular webpage can readily launch puppetnet attacks; many of the top-500 sites are highly suspect offering "warez" and other illegal downloads. Furthermore, we have found that some well-known underground sites, not listed in the top-500, can create puppetnets of 10,000-70,000 puppets (see [Lam et al. 2006]). Finally, the authors of reference [Wang et al. 2006] report that by scanning the most popular one million webpages according to a popular search engine, they found 470 malicious sites, many of which serve popular content related to celebrities, song lyrics, wallpapers, video game cheats, and wrestling. These malicious sites were found to be luring unsuspected users with the purpose of installing malware on their machines by exploiting client-side vulnerabilities. The compromised machines are often used to form a botnet, but visits to these popular sites could be used for staging a puppetnet attack instead.

Another way to stage a puppetnet attack is by compromising and injecting puppetnet code to a popular website. Although one might argue that popular sites are more likely to be secure, checking the top-500 sites from Alexa against the defacement statistics from zone-h[zone-h 2006] reveals that in the first four months of 2006 alone, 7 pages having the same domain as popular sites were defaced. For the entire year 2005 this number reaches 18. We must note, however, that the defaced pages were usually not front pages, and therefore their hits are likely to be less than those of the front pages. We also found many

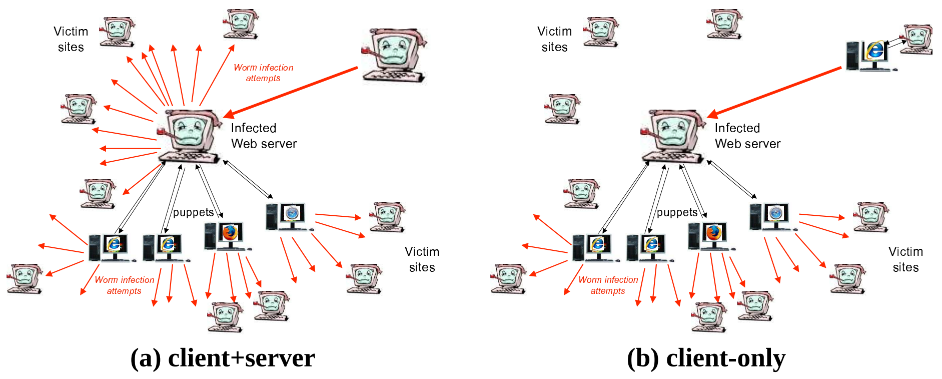


Fig. 7. **Two different ways that puppetnets could be used for worm propagation: (a) illustrates an infected server that uses puppets to propagate the worm, and (b) a server that propagates only through the puppet browsers.**

of them running old versions of Apache and IIS, although we did not go as far as running penetration tests on them to determine whether they were patched or not.

2.2 Worm propagation

Puppetnets can be used to spread worms that target vulnerable websites through URL-encoded exploits. Vulnerabilities in Web applications are an attractive target for puppetnets as these attacks can usually be encoded in a URL and embedded in a webpage. Web applications such as blogs, wikis, and bulletin boards are now among the most common targets of malicious activity captured by honeynets. The most commonly targeted applications according to recent statistics [Philippine Honeynet Project] are Awstats, XMLRPC, PHPBB, Mambo, WebCalendar, and PostNuke.

A Web-based worm can enhance its propagation with puppetnets as follows. When a web server becomes infected, the worm adds puppetnet code to some or all of the webpages on the server. The appearance of the pages could remain intact, just like in our DDoS attack, and each unsuspected visitor accessing the infected site would automatically run the worm propagation code. In an analogy to real-world diseases, web servers are *hosts* of the worm while browsers are *carriers* which participate in the propagation process although they are not vulnerable themselves. Besides using browsers to increase the aggregate scanning rate, a worm could spread entirely through browsers. This could be particularly useful if behavioral blockers prevent servers from initiating outbound connections. Furthermore, puppetnets could help worms penetrate NAT and firewall boundaries, thereby extending the reach of the infection to networks that would otherwise be immune to the attack. For example, the infected web server could instruct puppets to try propagating on private addresses such as 192.168.x.y. The scenarios for worm propagation are shown in Figure 7.

2.2.1 *Analysis of worm propagation.* To understand the factors affecting puppetnet worm propagation we first utilize an analytical model, and then proceed to measure key parameters of puppetnet worms and use simulation to validate the model and explore the resulting parameter space.

Analytical model: We have developed an epidemiological model for worm propagating using puppetnets. Briefly, we have extended classical homogeneous models to account for how clients and servers contribute to worm propagation in a puppetnet scenario. The key parameters of the model are the browser scanning rate, the puppetnet size, and the time spent by puppets on the worm-spreading webpage.

2.2.2 *Modeling puppetnet worm propagation.* To investigate the propagation dynamics of a puppetnet worm, we adopt the classical Random Constant Spread (RCS) model, as used in other studies [Staniford et al. 2002]. We use a notation system similar to [Staniford et al. 2002]:

—N: vulnerable population
—a: fraction of vulnerable machines which have been compromised
—a_0: initial fraction of compromised machines
—t: time
—t_h: mean holding time/latency of users (e.g., the amount of time the browser is actively displaying the infected Web page)
—C: number of concurrent clients per server
—K_s: server's initial compromise rate
—K_c: client's initial compromise rate
—K_p: puppetnet's initial compromise rate
—K: overall initial compromise rate
—$\psi(t)$: distribution process of holding time on a web page

At time t, the number of new machines being compromised within dt is affected by two parameters:

—Due to server: $Na(t)K_s(1-a(t))dt$.
—Due to puppetnet: each server has C clients and every client compromises at a rate of $K_c(1-a)$. However, due to the time $\psi(t)$ between web page infection and the user browsing to another infected page on the server, only puppetnets controlled by servers that have become active at time t are capable of propagating worm.

$$Nda(t) = Na(t)K_s(1-a(t))dt + N\left[\int_0^t a(\tau)\psi(t-\tau)d\tau\right] CK_c(1-a(t))dt$$

where $a(t)$ and $\psi(t)$ express full notation of a and ψ as functions of time t. Note that $a(t) = 0$ and $\psi(t) = 0$ for $t < 0$.

$$\frac{da}{dt} = K_s a(1-a) + \left[\int_0^t a(\tau)\psi(t-\tau)d\tau\right]CK_c(1-a) \qquad (1)$$

Equation 1 is generally non-linear without closed form solution and therefore could only be solved numerically.

We consider two cases of $\psi(t)$ as shown in Figure 8 and Figure 9. Figure 8 shows a model of a user's latency process at a web server. Clients slowly arrive at the web server at linear rate before and after a mean holding time t_h, until all users are either new users that joined after the server was infected, or old users that refreshed the page (and hence got

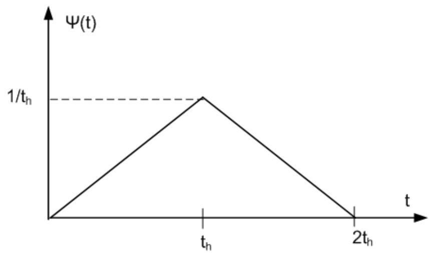


Fig. 8. Triangular latency process of clients

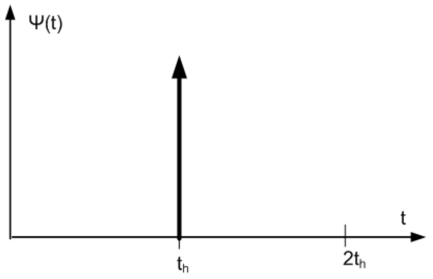


Fig. 9. Impulse arrival of clients : universal holding time t_h

the infected one) or browsed to another page on the same server. There is no closed-form solution for Equation 1 in this case.

Note that the area bounded by $\psi(t)$ and x-axis is unity: $\int_0^\infty \psi(\tau)d\tau = 0$

Figure 9 is a further simplification, with the holding time becoming a universal constant t_h. This means that all puppetnets controlled by servers that are infected at time $(t - t_h)$ become active at time t. With this assumption, $\psi(t)$ is an impulse function at $t = t_h$. Despite its simplicity, a symbolic approximate solution could be found, which gives us some intuition on the role of different parameters on the worm outbreak.

$[\int_0^t a(\tau)\psi(t-\tau)d\tau] = a(t-t_h) \approx a(t) - t_h\frac{da(t)}{dt}$ (first order approximation when t is sufficiently large).

Substituting in Equation 1, we get:

$$\frac{a^{1+K_p t_h}}{1-a} = e^{K(t-T)}$$

where K_p = CK_c and K = K_s + K_p

—If t_h is negligible, the final solution form becomes:

$$a = \frac{e^{K(t-T)}}{1+e^{K(t-T)}}$$

where T is a constant that specifies the time of the initial worm outbreak. The propagation pattern is exactly the same as a normal worm without puppetnets [Staniford et al. 2002], except that the worm propagation rate is increased by the contribution of puppets: K = K_s + K_p. More importantly, this contribution is significant if $K_p >> K_s$.

—If $t_h > 0$, only a numerical solution can be obtained. Compared to the above case of zero holding time, the worm outbreak is delayed by:

$$\Delta t = \frac{K_p}{K} t_h ln \frac{a}{a_0}$$

In all cases, worm propagation with puppetnets obeys the logistic form: its graph has the same sigmoid shape as a typical random scanning worm. The contribution to the propagation process by clients and servers mainly depends on the constants K_s (sever-side) and K_c (client-side).

Finally, we examine puppetnet-like viral propagation characteristics for general computer viruses and worms. We incorporate puppetnet factor into two general epidemiological models: Susceptible-Infected-Susceptible (SIS) and Susceptible-Infected-Removed (SIR) [Wang and Wang 2003].

In this case, δ is defined as the node cure rate (or virus death rate), i.e. the rate at which a node will be cured if it is infected. Note that our modified-RCS model is a special case of the SIS model with $\delta = 0$.

The modified SIS equation for puppetnet worms is:

$$\frac{da}{dt} = K_s a(1-a) + \big[\int_0^t a(\tau) e^{-\delta(t-\tau)} \psi(t-\tau) d\tau\big] C K_c (1-a) - \delta a \qquad (2)$$

while the steady state solution is:

$$a(\infty)_{SIS} = 1 - \frac{\delta}{K_s + CK_c \int_0^\infty \frac{\psi(\tau)}{e^{\delta\tau}} d\tau}$$

Additionally, the modified SIR equation for puppetnet worms is:

$$\frac{da}{dt} = K_s a(1-a-\delta\int_0^t a(\tau)d\tau) + \big[\int_0^t a(\tau) e^{-\delta(t-\tau)} \psi(t-\tau) d\tau\big] C K_c (1-a-\delta\int_0^t a(\tau)d\tau) - \delta a \qquad (3)$$

and the steady state solution for this case is:

$$a(\infty)_{SIR} = \frac{\big[K_s + CK_c \int_0^\infty \frac{\psi(\tau)}{e^{\delta\tau}} d\tau\big] - \delta}{\big[K_s + CK_c \int_0^\infty \frac{\psi(\tau)}{e^{\delta\tau}} d\tau\big]\big(1 + \delta.\infty\big)} = 0$$

Therefore, with the puppetnet enhancement, the final epidemic state for the SIR model is also zero.

2.2.3 *Model application.*

Scanning performance: If the attacker relies on simple web requests, the scanning rate is constrained by the default browser connection timeout and limits imposed by the OS and the browser on the maximum number of outstanding connections. In our proof-of-concept attack, we have embedded a hidden HTML frame with image elements into a normal webpage, with each image element pointing to a random IP address with a request for the attack URL. Note that the timeout for each round of infection attempts can be much lower than the time needed to infect all possible targets (based on RTTs). We assume that the redundancy of the worm will ensure that any potential miss from one source is likely to be within reach from another source.

Experimentally, we have found that both MSIE and Firefox on an XP SP2 platform can achieve maximum worm scanning rates of roughly 60 scans/min, mostly due to OS connection limiting. On other platforms, such as Linux, we found that a browser can perform roughly 600 scans/min without noticeable impact on regular activities of the user. These measurements were consistent across different hardware platforms and network connections.

Impact on worm propagation: We simulate a puppetnet worm outbreak and compare results with the prediction of our analytical model. We consider CodeRed [CERT 2001a] as an example of a worm targeting web servers and use its parameters for our experiments. To simplify the analysis, we ignore possible human intervention such as patching, quarantine, and the potential effect of congestion resulting from worm activity.

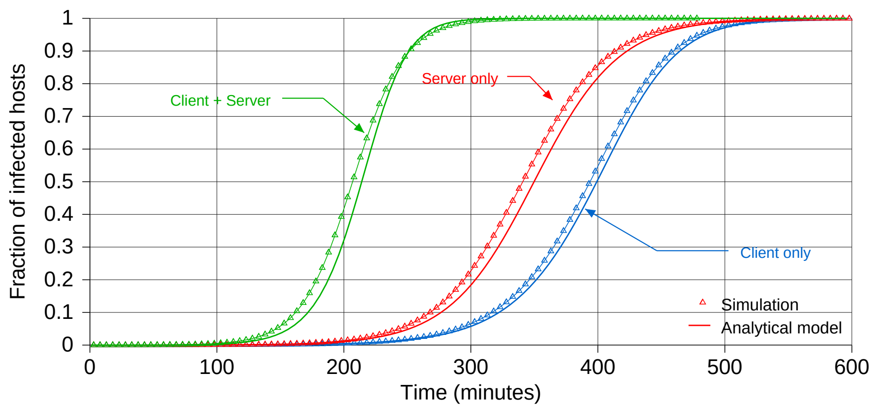
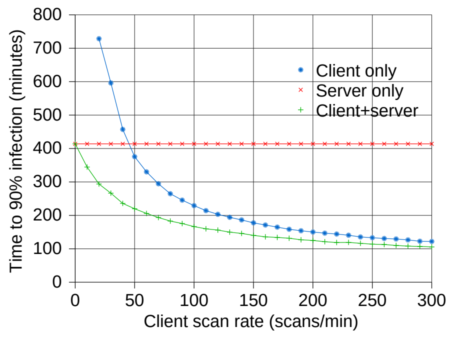


Fig. 10. **Worm propagation with puppetnet**



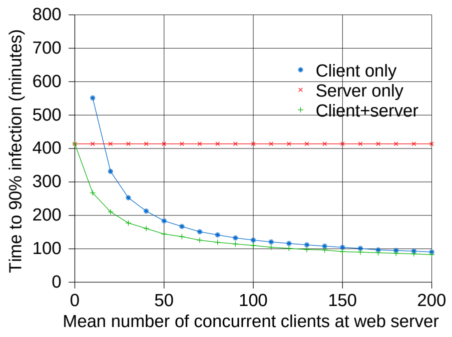


Fig. 11. **Worm infection for different browser scan rates**

Fig. 12. **Worm infection versus popularity of web servers**

We examine three different scenarios: (a) a normal worm where only compromised servers can scan and infect other servers, (b) a puppetnet-enhanced worm where both the compromised servers and their browsers propagate the infection, and (c) a puppetnet-only worm where servers only push the worm solely through puppets to achieve stealth or bypass defenses.

We have extended a publicly available CodeRed simulator [Zou et al. 2002] to simulate puppetnet worms. We adopt the parameters of CodeRed as used in [Zou et al. 2002]: a vulnerable population of 360,000 and a server scanning rate of 358 scans/min. In the simulation, we directly use these parameters, while in our analytical model, we map these parameters to analytical model parameters and numerically solve the differential equations. Note that our model is a special case of the Kephart-White Susceptible-Infected-Susceptible (SIS) model [Kephart and White 1991] with no virus curing. The compromise rate is $K = \beta \times \langle k \rangle$ where β is the virus birth rate defined on every directed edge from an infected node to its neighbors, and $\langle k \rangle$ is the average node out-degree. Assuming the Internet is a fully connected network, $\langle k \rangle_{CodeRed} = 360,000$ and $\beta_{CodeRed,server} = 358/2^{32}$, we have $K_s = 0.03$. Our simulation and analytical model also include the delay in accessing a webpage as users have to click or reload a newly-infected webpage to start participating in worm propagation.

We obtain our simulation results by taking the mean over five independent runs. For this experiment, we use typical parameters measured experimentally: browsers performing 36 scans/min (*i.e.,* an order of magnitude slower than servers), and web servers with about 13 concurrent users, and an average page holding time of 15 minutes. To study the effect of these parameters, we vary their values and estimate worm infection time as shown in Figures 11 and 12.

Figure 10 illustrates the progress of the infection over time for the three scenarios. In all cases the propagation process obeys the standard S-shaped logistic growth model. The simulated virus propagation matches reasonably well with the analytical model. Both agree on a worm propagation time of 50 minutes for holding times in the order of $t_h = 15min$ (that is, compared to the case of zero holding time). A client-only worm can perform as well as a normal worm, suggesting that puppetnets are quite effective at propagating worm epidemics.

Figure 11 illustrates the time needed to infect 90% of the vulnerable population for different browser scanning rates. When browsers scan at 45 scans/min, the client-only scenario is roughly equivalent to the server-only scenario. At the maximum scan rate of this experiment (which is far more than the scan rate for MSIE, but only a third of the scan rate for Linux), a puppetnet can infect 90% of the vulnerable population within 19 minutes. This is in line with Warhol worms and an order of magnitude faster than CodeRed.

Figure 12 confirms that the popularity of compromised servers plays an important role in worm performance. The break-even point between the server-only and client-only cases is when web servers have 16 concurrent clients on average. For large browser scanning rate or highly popular compromised servers, the client-only scenario converges to the client-server scenario. That means that infection attempts launched from browsers are so powerful that they dominate the infection process.

Finally, in a separate experiment we found that if the worm uses a small initial hitlist to specifically target busy web servers with more than 150 concurrent visitors, the infection time is reduced to less than two minutes, similar to flash worms [Staniford et al. 2004].

2.3 Reconnaissance probes

We discuss how malicious or subverted websites can orchestrate distributed reconnaissance probes. Such probes can be useful for the attacker to locate potential targets before launching an actual attack. The attacker can thereby operate in stealth, rather than risk triggering detectors that look for aggressive opportunistic attacks. Furthermore, as in worm propagation, puppets can be used to scan behind firewalls, NATs and detection systems. Finally, probes may also enable attackers to build *hitlists* that have been shown to result in extremely fast-spreading worms [Staniford et al. 2004].

As with DDoS, the attacker installs a webpage on the website that contains a hidden HTML frame that performs all the attack-related activities. The security model of modern browsers imposes restrictions on how the attacker can set up probing. For instance, it is not possible to ask the browser to request an object from a remote server and then forward the response back to the attacker's website. This is because of the so-called "same domain" (or "same origin") policy [Ruderman 2001], which is designed to prevent actions such as stealing passwords and monitoring user activity. For the same reason, browsers refuse access to the contents of an inline frame, unless the source of the frame is in the same domain with the parent page.

Unfortunately, there are workarounds for the attacker to indirectly infer whether a connection to a remote host is successful. The basic idea is similar to the timing attack of [Felten and Schneider 2000]. We “sandwich” the probe request between two requests to the attacker’s website. For example:

```
<IMG SRC='http://www.attacker.com/cgi-bin/ping'>
<IMG SRC='http://www.targetsite.com'>
<IMG SRC='http://www.attacker.com/cgi-bin/ping'>
```

We can infer whether the target is responding to a puppet by measuring the time difference between the first and third request. If the target does not respond, the difference will be either very small (*e.g.*, because of an ICMP UNREACHABLE message) or very close to the browser request timeout. If the target is responsive, then the difference will vary but is unlikely to coincide with the timeout.

Because browsers can launch multiple connections in parallel, the attacker needs to serialize the three requests. This can be done with additional requests to the attacker’s website in order to consume all but one connection slots. However, this would require both discovering and also keeping track of the available connection slots on each browser, making the technique complex and error-prone. A more attractive solution is to employ JavaScript, as modern browsers provide hooks for a default action after a page is loaded (the *onLoad* handler) and when a request has failed (the *onError* handler). Using these controls, the attacker can easily chain requests to achieve serialization without the complexity of the previous technique. We therefore view the basic sandwich attack as a backup strategy in case JavaScript is disabled.

We have tested this attack scenario as shown in Figure 13. In a hidden frame, we load a page containing several image elements. The script points the source of each image to the reconnaissance target. Setting the source of an image element is an asynchronous operation. That is, after we set the source of an image element, the browser issues the request in a separate thread. Therefore, the requests to the various scan targets start at roughly the same time. After the source of each image is set, we wait for a timeout to be processed through the *onLoad* and *onError* handlers for every image. We identify the three cases (*e.g.*, unreachable, live, or non-responsive) similar to the sandwich attack but instead of issuing a request back to the malicious website to record the second timestamp we collect the results through the *onLoad/onError* handlers.

After the timeout expires, the results can be reported to the attacker, by means such as embedding timing data and IP addresses in a URL. The script can then proceed to another round of scanning. Each round takes time roughly equal to the timeout, which is normally controlled by the OS. It is possible for the attacker to use a smaller timeout through the *setTimeout()* primitive, which speeds up scanning at the expense of false negatives. We discuss this trade-off in Section 2.3.1.

There are both OS and browser restrictions on the number of parallel scans. On XP/SP2, the OS enforces a limit of no more than ten “outstanding”[6] connection requests at any given time [Andersen and Abella 2004]. Some browsers also impose limits on the number

[6] A connection is characterized outstanding when the SYN packet has been sent but no SYN+ACK has been received.

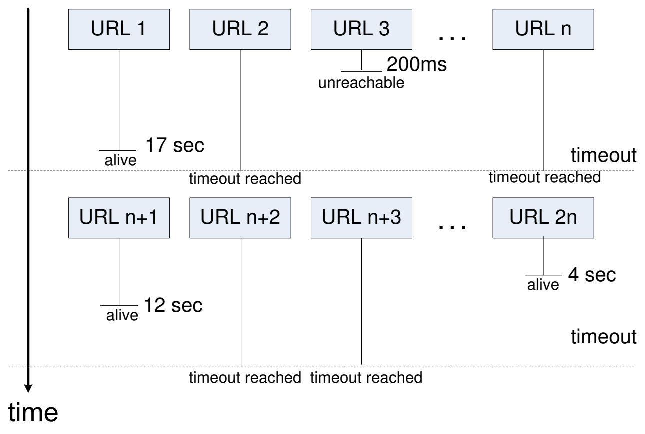


Fig. 13. **Illustration of reconnaissance probing method.**

of simultaneous established connections. MSIE and Opera on Windows (without SP2), and browsers such as Konqueror on Linux, impose no limits, while Firefox does not allow more than 24. The attacker can choose between using a safe common-denominator value or employing JavaScript to identify the OS and browser before deciding on the number of parallel scans. The code for scanning is shown in Figure 14.

We also considered the case where JavaScript is disabled. In that case, the effectiveness of scanning is constrained by the maximum number of parallel connections allowed by the browser and the operating system. With JavaScript-disabled, the attacker has to revert to the sandwich attack and make assumptions or take measurements to determine the timeout values and maximum connections for different browsers and operating systems. Moreover, the attacker cannot cancel the timeout and move on to the next request as is the case with JavaScript. We have experimentally found that for both Windows and Linux systems, a web connection times out after 22 seconds. That is we can have roughly 3 scanning sessions per minute. A simplifying assumption for the attacker would be to assume that all users have XP/SP2 installed on and that all connection slots in the browser are free. As the number of outstanding connections is limited by the operating system to 10, then the attacker is limited to scan for around 30 hosts per minute. Without SP2, this number increases up to 72, as attacker has 24 available slots to search for hosts.

The same process can be used to identify services other than web servers. When connecting to such a service, the browser will issue an HTTP request as usual. If the remote server responds with an error message and closes the connection, then the resulting behavior is the same as probing web servers. This is the case for many services, including SSH: the SSH daemon will respond with an error message that is non-HTTP-compliant and cannot be understood by the browser, the browser will subsequently display an error page, but the timing information is still relevant for reconnaissance purposes. This approach, however, does not work for all services, as some browsers block certain ports: MSIE blocks ports FTP, SMTP, POP3, NNTP and IMAP to prevent spamming through webpages (we return to this problem in Section 2.4); Firefox blocks a larger number of ports [ffb 2004]; interestingly, Apple's Safari does not impose any restrictions. The attacker can rely on the "User-agent" string to trigger browser-specific code.

```
<script language="javascript">
function printTimes() {
    var i=0;
    var curdate = new Date();
    var start_time= curdate.getTime();

    for(i=0;i<parallel_scans;i++) {
        if(loadTime[i]>0)
          info = info+"url="+victim[i]+",time="+loadTime[i]+"\n";
        victim[i]="http://"+createRandomIP();
        document.getElementById("frame"+i).src=victim[i];
        loadTime[i]=0;
        scanStarted[i]=start_time;
    }
    setTimeout('printTimes()',20000);
}

function startScanning() {
    var i=0;
    init();
    var curdate = new Date();
    start_time= curdate.getTime();

    for(i=0;i<parallel_scans;i++) {
       scanStarted[i]=start_time;
       /* stopTimer calculates the difference between the time
       we issued the request and the time requested URL was
       loaded or we had an error (not found) */
       document.write("<img id=frame"+i+" src=\""+victim[i]+"\"
       onload=\"stopTimer("+i+")\"
       onerror=\"stopTimer("+i+")\"></img>");
    }
    setTimeout('printTimes()',20000);
}

</script>
```

Fig. 14. **Scanning for liveness of hosts through puppets**

As described so far, puppetnets are limited to determining only the *liveness* of a remote target. As the same-domain policy restricts the attack to timing information only, the attack script cannot relay back to the attacker information on server software, OS, protocol versions, *etc.*, which are often desirable. Although this is a major limitation, distributed liveness scans can be highly valuable to an attacker. An attacker could use a puppetnet to evade detectors that are on the lookout for excessive numbers of failed connections, and then use a smaller set of sources to obtain more detailed information about each live target. Recent work by Bortz et al. [Bortz et al. 2007] has also shown that timing attacks can extract private information such as if a user is logged in at a certain site or way to learn the number of objects in the user's shopping cart. The authors of [Bortz et al. 2007] use a similar technique based on JavaScript timing functions, as it is described in the following

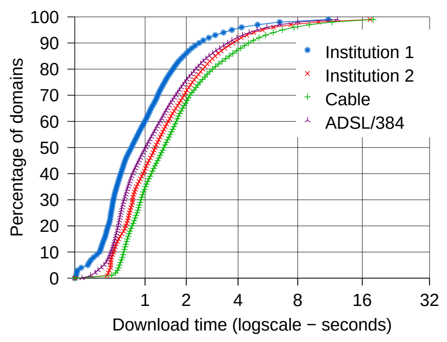


Fig. 15. **Time to get main index from different sites.**

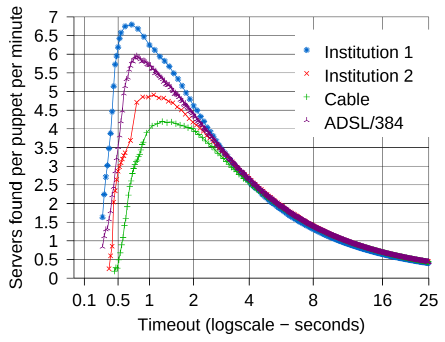


Fig. 16. **Discovery rate, per puppet.**

section.

2.3.1 *Analysis of reconnaissance probing.* There is a subtle difference between worm scanning and reconnaissance scanning. In worm scanning, the attacker can opportunistically launch probes and does not need to observe the result of each probe. In contrast, reconnaissance requires data collection and reporting.

There are two parameters in the reconnaissance attack that we need to explore experimentally: the timeout for considering a host non-responsive, and the threshold for considering a host unreachable. The attacker can tune the timeout and trade off accuracy for speed of data collection. The unreachable threshold does not affect scanning speed, but if it is too large it may affect accuracy, as it would be difficult to distinguish between unreachable hosts and live hosts. Both parameters depend on the properties of the network through which the browser is performing the scans.

In our first experiment we examine how the choice of timeout affects reconnaissance speed and accuracy and whether the unreachable threshold may interfere with reconnaissance accuracy. As most browsers under the attacker's control are expected to stay on the malicious page only for a few minutes, the attacker may want to maximize the scanning rate. If the timeout is set too low, the attacker will not be able to discover hosts that would not respond within the timeout.

Note that in the case of XP/SP2, the timeout must be higher than the default connection timeout of the OS, which is 25 seconds. The reason is that the scanning process has to wait until outstanding connections of the previous round are cleared before issuing new ones. The analysis below is therefore more relevant to non-XP/SP2 browsers. We succeeded, in part, in circumventing the XP/SP2 connection limit by having the browser "refresh" the scan page. This caused connection state to be released, presumably because of a request to close the socket. Oddly, this workaround only worked for small numbers of parallel connections. At this point we do not fully understand how the connection limiter operates, and whether the workaround can be perfected.

We measure the time needed to download the main index file for roughly 50,000 unique websites, obtained through random queries to a search engine. We perform our measurements from four hosts in locations with different network characteristics. The distributions of the download times are presented in Figure 15. We see that in all cases, a threshold of 200-300 msec would result in a loss of around 5% of the live targets, presumably those within very short RTT distance from the scanning browser. We consider this loss to be acceptable.

Recall that the goal of the attacker may be speed rather than efficiency. That is, the attacker may not be interested in finding all servers, but finding a subset of them very quickly. We use simulation, driven by our measured distributions, to determine the discovery rate for a puppet using different timeout values, assuming 200 msec as the unreachable threshold. The results are summarized in Figure 16. For the four locations in our study, the peak discovery rate differs in absolute value, and is maximized at different points, suggesting that obtaining optimal results would require effort to calibrate the timeout on each puppet. However, all sources seem to perform reasonably well for small timeouts of 1-2 seconds.

In our second experiment we look at a 2-day packet trace from our institution and try to understand the behavior of ICMP unreachable notifications. We identify roughly 23,000 distinct unreachable events. The RTTs for these notifications were between 5 msec and 18 seconds. Nearly 50% responded within 347 msec, which, considering the response times of Figure 15, would result in less than 5% false negatives if used as a threshold. The remaining 50% of unreachables that exceed the threshold will be falsely identified as live targets, but as reported in [Berk et al. 2003], only 6.36% of TCP connections on port 80 receive an ICMP unreachable as response. As a result, we expect around 3% of the scan results to be false positives.

2.4 Protocols other than HTTP

One limitation of puppetnets is that they are bound to the use of the HTTP protocol. This raises the question of whether any other protocols can be somehow “tunneled” on top of HTTP. This can be done, in some cases, using the approach of [Topf 2001; CERT 2001b]. Briefly, it is possible to craft HTML forms that embed messages to servers understanding other protocols. The browser is instructed to issue an HTTP POST request to the remote server. Although the request contains the standard HTTP POST preamble, the actual post data can be fully specified by the HTML form. Thus, if the server fails gracefully when processing the HTTP part of the request (*e.g.,* ignoring them, perhaps with an error message, but without terminating the session), all subsequent messages will be properly processed. Two additional constraints for the attack to work is that the protocol in question must be text-based (since the crafted request can only contain text) and asynchronous (since all messages have to be delivered in one pass).

In this scenario, SMTP tunneling is achieved by wrapping the SMTP dialogue in a HTTP POST request that is automatically triggered through a hidden frame on the malicious webpage. For IRC servers that do not require early handshaking with the user (*e.g.,* the *identd* response), a browser can be instructed to login, join IRC channels and even send customized messages to the channel or private messages to pre-selected list of users sitting in that channel. This feature enables the attacker to use puppetnet for certain attacks such as triggering botnets, flooding and social engineering. The method is pretty similar to SMTP. An example of how a web server could instruct puppets to send spam is provided in [Lam et al. 2006].

Although this vulnerability has been discussed previously, its potential impact in light of a puppetnet-like attack infrastructure has not been considered, and vendors may not be aware of the implications of the underlying vulnerability. We have found that although MSIE refuses outgoing requests to a small set of ports (including standard ports for SMTP, NNTP, *etc.*) and Firefox blocks a more extensive list of ports, Apple’s Safari browser

```
<FORM METHOD="POST" NAME="myform"
      onSubmit="return nextjob()"
      enctype="multipart/form-data"
      ACTION="http://mailserver:25/foo">
<TEXTAREA name="thetext" rows="0" cols="0">

MAIL FROM: <spammer@marketing.com>
RCPT TO: <victim@target.com>
DATA
Subject: viagra

ok lah
.
QUIT

</TEXTAREA>
</FORM>
<body onLoad="document.myform.submit()">
```

Fig. 17. **Spam transmission through puppets**

as well as MSIE5.2 on Mac OSX do not impose *any* similar port restrictions[7]. Thus, although the extent of the threat may not be as significant as DDoS and worm propagation, popular websites with a large Apple/Safari user base can be easily turned into powerful spam conduits. The example code for sending spam can be seen at Figure 17.

2.5 Exploiting cookie-authenticated services

A large number of Web-based services rely on cookies for maintaining authentication state. A typical example is Web-based mail services that offer a “remember me” option to allow return visits without re-authentication. Such services could be manipulated by a malicious site that coerces visitors to post forms created by the attacker with the visitors’ credentials. There are three constraints for this attack. First, the inline frame needs to be able to post cookies; this works on Firefox, but not MSIE. Second, the attacker needs to have knowledge about the structure and content of the form to be posted, as well as the target URL; this depends on the site design. Finally, the attacker needs to be able to instruct browsers to automatically post such forms; this is possible in all browsers we tested.

We have identified sites that are vulnerable to this attack.[8] As proof-of-concept, we have successfully launched an attack to one of our own accounts on such a site. Although this seems like a wider problem (*e.g.,* it allows the attacker to forward the victim’s email to his site, *etc.*), in the context of puppetnets, the attacker could be on the lookout for visitors that happen to be pre-authenticated to one of the vulnerable websites, and could use them for purposes such as sending spam or performing mailbomb-type DoS attacks. Once again, puppetnets here only amplify an existing flaw that in many scenarios may be regarded as one of minor importance, as it may not be potent enough to cause significant damage in isolated incidents. However, in some cases this flaw can be a serious concern as it enables

[7]We have informed Apple about this vulnerability.
[8]These sites include a very popular Web-based mail service, the name of which we would prefer to disclose only upon request.

targeted attacks. The most common defense to this kind of attack is to require session-specific state in any form submission request which is only known to the relevant browser windows or frames, or to perform strict referrer checking on the server side.

Given the restriction to Firefox and the need to identify visitors that are pre-authenticated to particular sites, it seems that this attack would only have significant impact on highly popular sites, or moderately popular sites with unusually high session times, or sites that happen to have an unusually large fraction of Firefox visitors. Considering these constraints, the attack may seem weak compared to the ubiquitous applicability of DoS, scanning, and worm propagation. Nevertheless, none of these three scenarios can be safely dismissed as unlikely.

2.6 Distributed malicious computations

So far we have described scenarios of puppetnets involved in network-centric attacks. However, besides network-centric attacks, it is easy to imagine browsers unwillingly participating in malicious computations. This is a form of Web-based computing which, to the best of our knowledge, has not been considered as a platform for malicious activity. Projects such as RC5 cracking [Gladychev et al. 1998], use the Web as a platform for distributed computation but this is done with the users' consent. Most large-scale distributed computing projects rely on stand-alone clients, similar to SETI@home [Korpela et al. 2001]. This particular form of abuse differs from the attacks we described previously as it does not directly involve browser resources being abused to attack a third-party, as the victim may be "offline". Furthermore, there is no well-defined policy as to what is legitimate and what may be harmful in terms of computation – this may well be subjective, and there may be no way to ensure that CPU cycles are used for benign purposes. This makes it difficult to defend against such abuse, but also illustrates a larger point regarding puppetnets, in that they involve *amplification* of threats through large-scale abuse of browser resources.

It is easy to instruct a browser to perform local computations and send the results back to the attacker. Computation can be done through JavaScript, Active-X or Java applets. By default, Active-X does not appear attractive as it requires user confirmation. JavaScript offers more stealth as it is lightweight and can be made invisible. Sneaking Java applets into hidden frames on malicious websites seems easy, and although the resources needed for instantiating the Java VM might be noticeable (and an "Applet loaded" message may be displayed on the status bar), it is unlikely to be considered suspect by a normal user.

To illustrate the extent of the problem we measured the performance of JavaScript and Java applets for MD5 computations. On a low-end desktop, the JavaScript implementation can perform around 380 checksums/sec, while the Java applet within the browser can compute roughly 434K checksums/sec – three orders of magnitude faster than JavaScript. Standalone Java can achieve up to 640K checks/sec. In comparison, an optimized C implementation computes around 3.3M checks/sec. Hence, a 1,000-node puppetnet can crack an MD5 hash as fast as a 128-node cluster.

3. DEFENSES

In this section we examine potential defenses against puppetnets. The goal is to determine whether it is feasible to address the threat by tackling the source of the problem, rather than relying on techniques that attempt to mitigate the resulting attacks, such as DDoS, which may be hard to implement right at a global scale.

We discuss various defense strategies and the tradeoffs they offer. We concentrate on defenses against DDoS, scanning and worm propagation. Detecting malicious computations seems hard, and well beyond the scope of this paper. Cookie-authenticated services seem trivial to protect by adding non-cookie session state that is communicated to the browser when the user wishes to re-authenticate.

Disabling JavaScript. The usual culprit, when it comes to Web security problems, is JavaScript, and it is often suggested that many problems would go away if users disable JavaScript and/or websites refrain from using it. However, the trade-off between quality content and security seems unfavorable: the majority of websites employ JavaScript, there is growing demand for feature-rich content, especially in conjunction with technologies such as Ajax[Garrett 2005], and most browsers are shipped with JavaScript enabled. It is interesting to note, however, that a recently-published Firefox extension that selectively enables JavaScript only for “trusted” sites [Maone 2006] has been downloaded 7 million times roughly one month after its release on April 9th, 2006.

In the case of puppetnets, disabling JavaScript could indeed alter the threat landscape, but it would only reduce rather than eliminate the threat. The development of our attacks suggests that even without JavaScript, it would still be feasible to launch DDoS, perform reconnaissance probes and propagate worms. We have found that DDoS is almost as potent without JavaScript. On the other hand, scanning and worm propagation are much slower and more complicated. Considering these observations, disabling JavaScript does not seem like an attractive proposition towards completely eliminating the puppetnet threat.

Careful implementation of existing defenses. We observe that in most cases the attacks we developed were quite sensitive to minor tweaks. That is, although simple versions of the attack were quite easy to construct, maximizing their effectiveness required a lot more effort. Particularly the connection rate limiter implemented in XP/SP2 had a profound effect on the performance of worm propagation and reconnaissance. Unfortunately, we were able to demonstrate that the rate limiter can be partially bypassed. In particular, by reloading the core attack frame we were able to clear the TCP connection cache, presumably because a frame reload results in the underlying socket being closed and the TCP connection state entry being removed.

In this particular case it appears easy to address the problem by means of separating the actual connection state table from the state of the connection rate limiter. The rate limiter could either mirror the regular connection state table but choose to retain entries for closed sockets up to a timeout, or keep track of aggregate connection state statistics. This could reduce the effectiveness of worm propagation of up to an order of magnitude.

Another case that suggests that existing defenses are not always properly implemented is the Spam distribution attack described in Section 2.4. Although both MSIE and Firefox have mitigated this problem, at least in part, through blocking certain ports, Apple’s Safari and the OSX version of MSIE5.2 did not properly address this known vulnerability.

However, careful implementation of existing defenses is insufficient for addressing the whole range of threats posed by puppetnets.

Filtering using attack signatures. We consider whether it is practical to develop IDS/IPS signatures for puppetnet attacks. In some cases it seems fairly easy to construct such a signature. For example, in the case of puppetnet-delivered spam it is easy to scan traffic for messages to the SMTP port that contain evidence of both a HTTP POST request *and*

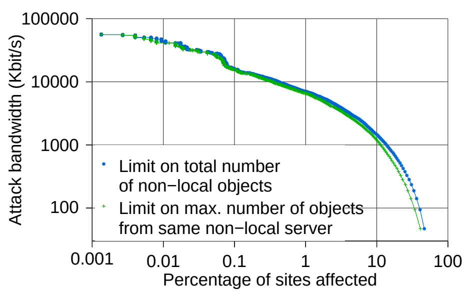


Fig. 18. **Effectiveness of remote request limits**

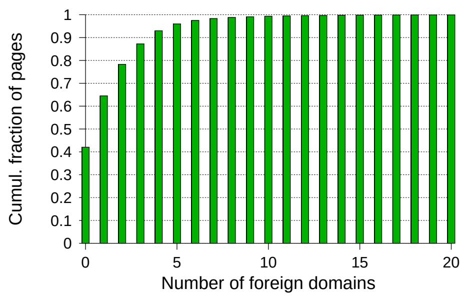


Fig. 19. **Cumul. histogram of foreign domains referenced by websites**

legitimate SMTP commands. This should cover most other protocols tunneled through POST requests.

Can we develop signatures for puppetnet DoS attacks? We could consider signatures of malicious webpages that contain unusually high numbers of requests to third-party sites. However, our example attack suggests that there are many possible variations to the attack, making it hard to obtain a complete set of signatures. Additionally, because the attacks are implemented in HTML and JavaScript, it appears unlikely that simple string matching or even regular expressions would be sufficient for expressing the attack signatures. Instead, more expensive analyzers, such as HTML parsers, would be needed.

Furthermore, obfuscation of HTML and JavaScript seems to be both feasible and effective [Stunnix 2006; Ross et al. 2005], allowing the attacker to compose obfuscated malicious webpage on-the-fly. For example, one could use the *document.write()* method to write the malicious page into an array in completely random order before execution. This makes the attack difficult to detect using static analysis alone, a problem that is also found in shellcode polymorphism [Chinchani and Berg 2005; Kruegel et al. 2005; Polychronakis et al. 2006]. Although we must leave room for the possibility that such a unusual use of *document.write()* or similar approaches may be amenable to detection, such analysis seems complex and is likely to be expensive and error-prone.

Client-side behavioral controls. Another possible defense strategy is to further restrict browser policies for accessing remote sites. It seems relatively easy to have a client-side solution deployed with a browser update, as browser developers seem to be concerned about security, and the large majority of users rely on one among 2-3 browsers.

One way to restrict DoS, scanning and worm propagation is to establish bounds on how a webpage can instruct the browser to access “foreign” objects, *e.g.,* objects that do not belong to the same domain. These resource bounds should be persistent, to prevent attackers from resetting the counters using page reloads. For similar reasons, the bounds should be tied to the requesting server, and not to a page or frame instance, to prevent attackers from evading the restriction through multiple frames, chained requests, *etc*.

We consider whether it makes sense to impose controls on foreign requests from a webpage. We attempt to quantify whether such a policy would break existing websites, and what impact it would have on DDoS and other attacks. We first look at whether we can limit the total number of different objects (*e.g.,* images, embedded elements and frames) that the attacker can load from foreign servers, considering all servers to be foreign except for the one offering the page. This restriction should be enforced across multiple automatic refreshes or frame updates, to prevent the attacker from resetting the counters upon reaching the limit. (Counters would only be reset only when a user clicks on a link.) Of course, this is likely to "break" sites such as those that use automatic refresh to update banner ads. Given that ads are loaded periodically, *e.g.*, one refresh every few minutes, it seems reasonable to further refine the basic policy with a timeout (or leaky bucket mechanism) that occasionally resets or tops-up the counters.

To evaluate the effectiveness of this policy, we have obtained data on over 70,000 webpages by crawling through a search engine. For each webpage we obtain the number of non-local embedded object references. We then compute for each upper bound N of non-local references, the fraction of sites that would be corrupted should such a policy be implemented, against the effective DoS bandwidth of a 1000-node puppetnet under the same policy. A variation of the above policy involves a cap on the maximum number of non-local references to the same non-local server.

The results are shown in Figure 18. We observe that this form of restriction is somewhat effective when compared to the DDoS firepower of Figure I, providing a 3-fold reduction in DDoS strength when trying to minimize disruption to websites. The strength of the attack, however, remains significant, at 50 Mbit/s for 1000 puppets. Obtaining a 10x reduction in DDoS strength would disrupt around 0.1% of all websites, with DDoS reduced to 10 Mbit/s. Obtaining a further 10x reduction seems impractical, as the necessary request cap would negatively affect more than 10% of webpages. The variation limiting the max. number of requests to the same non-local server also does not offer any notable improvement. Given the need to defend against not only 1000-node but even larger puppetnets, we conclude that although the improvement offered is significant, this defense is not good enough to address the DDoS threat.

The above policy only targets DDoS. To defend against worms and reconnaissance probes, we look at the feasibility of imposing limits on the number of distinct remote servers to which embedded object references are made. The cumulative histogram is shown in Figure 19. We see that most websites access very few foreign domains: around 99% of websites access 11 or less foreign domains; around 99.94% of websites access less than 20 foreign domains. Not visible on the graph is a handful of websites, typically "container" pages, that access up to 33 different domains. Based on this profile, it seems reasonable to implement a restriction of around 10 foreign domains, keeping in mind that the limit should be set as low as possible, given that a large fraction of puppets have a very short lifetime in the system. Note that sites that are out of profile could easily avoid violating the proposed policy, by proxying requests to the remote servers. We repeated the worm simulation of Section 2.2.1 to determine the impact of such a limit on worm propagation. As expected, this policy almost completely eliminates the speed-up for the client-server worm compared to server-only, as puppets can perform only a small fraction of the scans they could perform without this policy. Similar effects apply to scanning as well.

Unfortunately, the above heuristic can be circumvented if the attacker has access to a DNS server. The attacker could map all foreign target hosts to identifiers that appear to be in the same domain but are translated by the attacker's DNS server to IP addresses in the foreign domain. Attacks aiming at consuming egress bandwidth from servers that rely on the "Host:" tag in the HTTP request would be less effective, but all other attacks are not affected.

Server-side controls and puppetnet tracing. Considering the DoS problem and the difficulty in coming up with universally acceptable thresholds for browser-side connection limiting, one could argue that it is the Web developers who should specify how their sites are accessed by third parties.

One way for doing that is for servers to use the "Referer" tag of HTTP requests to determine whether a particular request is legitimate and compliant, similar to [Wang 2005]. The server could consult the appropriate access policy and decide whether to honor a request. This approach would protect servers against wasting their egress bandwidth, but does not allow the server to exercise any control over incoming traffic.

Another use of referrer information can be to trace the source of the puppetnet attack, and take action to shutdown the control website. That is, puppetnets have a single point of failure. However, this process is relatively slow as it involves human coordination. Thus, attackers may already have succeeded in disrupting service. Moreover, even when the controlling site has been taken down, existing puppets will continue to perform an attack – the attack will only subside once all puppet browsers have been pointed elsewhere, which is likely to be in the order of 10-60 minutes, based on the viewing time estimates of Section 2.1.1.

However, as shown in [Lam et al. 2006], we have been able to circumvent the default behavior of browsers that set referrer information, making puppetnet attacks more difficult to filter and trace. It is unclear at this point if minor modifications could address the loss of referrer-based defenses. Thus, referrer-based filtering does not currently offer much protection and may not be sufficient, even in the longer-term, for adequately protecting against puppetnet attacks.

Server-directed client-side controls. To protect against unauthorized incoming traffic from puppets, we examine the following approach. If we assume that the attacker cannot tamper with the browser software, a server can communicate site access policies to a browser during the first request. In our implementation, we embed Access Control Tokens (ACTs) in the server response through an extension header ("X-ACT:") that is either a blanket "permit/deny" or a JavaScript function, similar to proxy autoconfiguration[Mozilla.org 2004]. This script is executed on the browser side for each request to the server to determine whether a request is legitimate or not. The use of JavaScript offers flexibility for site developers to design their own policies, without having to standardize specific behaviors or a new policy language.

The scope of ACTs is a sensitive issue. If a server is able to specify ACTs for the IP address it is responding to, this gives any hosted server on that same IP control over all other hosted servers, opening up an opportunity for abuse. The hosting system would have to perform scope checks on ACTs or otherwise ensure that servers cannot abuse ACTs. If a webserver can only set policies for its own "domain" similar to the same-origin principle, then it may possible for a malicious or subverted server to create ACTs that result in denial-of-service on sub-domain servers. If this becomes an issue, it is straightforward to restrict the scope of ACTs to the same webserver only, at the expense of having to set separate ACTs for each server involved in a service (e.g., image server, etc.). Another problem with domain-level ACTs is that the attacker can still use aliasing as discussed previously to amplify his attack. The use of certificates binding IP addresses to server or domain names,

Distribution	**Firepower**
KDDCUP	47.03
Google/WT	14.39
JSTracker	3.05
Web track	2.87

Fig. 20. **Impact of ACT defense on 1000-puppet DDoS attack**

as we will discuss further in this section, is one way to address this problem.

Perhaps the simplest policy would be to ask browsers to completely stop issuing requests if the server is under attack. More fine-grained policies might restrict the total number or rate of requests in each session, or may impose custom restrictions based on target URL, referrer information, navigation path, etc. One could envision a tool for site owners to extract behavioral profiles from existing logs, and then turn these profiles into ACT policies. For a given policy, the owners can also compute the exposure in terms of potential puppetnet DDoS firepower, using the same methodology used in this paper. The specifics of profiling and exposure estimation are beyond the scope of this paper.

ACTs require at least one request-response pair for the defense to kick in, given that the browser may not have communicated with the server in the past. After the first request, any further unauthorized requests can be blocked on the browser side. Thus, ACTs can reduce the DoS attack strength to one request per puppet, which makes them quite attractive. On the other hand, this approach requires modifications to both servers and clients.

To illustrate the effectiveness of this approach, we estimate, using simulation, the firepower of a 1000-puppet DDoS attack where all users support ACTs on their browsers. The puppet viewing time on the malicious or subverted site is taken from the distributions shown in Figure 3. The victim site follows the most conservative policy: if a request comes from a non-trusted referrer then user is not allowed to make any further requests. The results are summarized in Table 20. As the attack is restricted to one request per user, the firepower is limited to only a few Kbit/sec.

In theory, it is possible to prevent the first unauthorized request to the target if policies are communicated to the browser out-of-band. One could directly embed ACTs in URL references, through means such as overloading the URL. Given that an ACT needs to be processed by the browser, it must fully specify the policy "by value". To prevent the attacker from tampering with the ACT, it must be cryptographically signed by the server. Besides being cumbersome, this also requires the browser to have a public key to verify the ACTs. While (most) SSL websites already have certificates in place, this proposal is less attractive for all other non-SSL websites.

Patrolling the Web for puppetnets. Another direction is to crawl the Internet for websites employing puppetnet attacks, similar to how honeymonkeys are used to discover other attacks [Wang et al. 2006]. To determine whether a website contains an attack, the honeymonkey system would have to be furnished with additional signatures and detection heuristics that are roughly equivalent to the techniques we described previously. Furthermore, it might be useful to launch multiple instances of the instrumented browser in order to capture the aggregate behavior of the puppetnet. Naturally, this approach inherits some of the deeper constraints of the patrol approach, such as providing a window of opportunity for the attacker between deployment of the attack and the next scan by the honeymonkey. Nevertheless, it might be worth considering as a short-term measure until other, more proactive defenses are widely deployed.

4. RELATED WORK

Web security has attracted a lot of attention in recent years, considering the popularity of the Web and the observed increase in malicious activity. Rubin *et al.* [Rubin and Jr. 1998] and Claessens *et al.* [Claessens et al. 2002] provide comprehensive surveys of problems and potential solutions in Web security, but do not discuss any third-party attacks like puppetnets. Similarly, most of the work on making the Web more secure focuses on protecting the browser and its user against attacks by malicious websites (c.f., [Kruegel and Vigna 2003; Ioannidis and Bellovin 2001; Felten et al. 1997; Chou et al. 2004; Jackson et al. 2006]).

The most well-known form of HTML tag misuse is known as cross-site scripting (or XSS) and is discussed in a CERT advisory in 2000 [CERT 2000]. The advisory focuses primarily on the threat of attackers injecting scripts into sites such as message boards, and the implications that such scripts could have on users browsing those sites, including potential privacy loss. Although XSS and puppetnet attacks both exploit weaknesses of the Web security architecture, there are two fundamental differences. First, puppetnet attacks require the attacker to have more control over a web server, in order to maximize exposure of users to the attack code. Injecting puppetnet code on message boards in a XSS fashion is also an option, but is less likely to be effective. The second important difference is that puppetnets exploit browsers for attacking third parties, rather than attacking the browser executing the malicious script. Several proposal for defending against such attacks have been recently explored (c.f., [Reis et al. 2006; Jim et al. 2007; Vogt et al. 2007]).

During the course of our investigation we became aware of a report [VNExpress 2005] describing a DDoS attack that appears to be very similar to the one described in this paper. The report, published in early December 2005, states that a well-known hacker site was attacked using a so-called “xflash” attack which involves a “secret banner” encoded on websites with large numbers of visitors redirecting users to the target. According to the same report, the attack generated 16,000 SYN packets per second towards the target. As we have not been able to obtain a sample of the attack code, we cannot directly compare it to the one described here. However, from the limited available technical information, it seems likely that attackers are already considering puppetnet-style techniques as part of their arsenal.

Another example of a puppetnet-like attack observed in the wild is so-called “referer spamming” [Healan 2003], where a malicious website floods some other site’s logs to make its way into top referer lists. The purpose of the attack is to trick search engines that rank sites based on link counts, since the victims will include the malicious sites in their top referer lists.

An attack similar to puppetnets is described in [Stamm et al. 2006]. That work was based on the observation that most users setup their home networks using inexpensive broadband users that offer wired and wireless networking capabilities. Malicious websites can trick users to open the configuration page of their home router and make them change network properties, especially the DNS server. That work focused primarily on how to mount pharming attacks that lead to denial of service, malware theft and identity theft.

The work that is most closely related to ours is a short paper by Alcorn [Alcorn 2005] discussing “XSS viruses”, developed independently and concurrently [Lam et al. 2005] to our investigation. The author of this work imagines attacks similar to ours, focusing on puppetnet-style worm propagation and also mentions the possibility of DDoS and spam distribution. The main difference is that our work offers a more in-depth analysis of each attack as well as concrete experimental assessment of the severity of the threat. For instance, a proof-of-concept implementation of an XSS virus that is similar to our puppetnet worm is provided albeit without analyzing its propagation characteristics. Similarly, DDoS and spam are mentioned as *potential* attacks but without any further investigation. The author discusses referer-based filtering as a potential defense, which, as we have shown, can be currently circumvented and is also unlikely to be sufficient in the long term. One major difference in the attack model is that we consider popular malicious or subverted websites as the primary vector for controlling puppetnets, while [Alcorn 2005] focuses on first infecting Web servers in order to launch other types of attacks. Similar ideas are also discussed in [Moniz and Moore 2006]. While the work of [Alcorn 2005] and [Moniz and Moore 2006] are both interesting and important, we believe that raising awareness and convincing the relevant parties to mobilize resources towards addressing a threat requires not just a sketch or proof-of-concept artifact of a potential attack, but extensive analysis and experimental evidence. In this direction, we hope that our work provides valuable input.

The technique we used for sending spam was first described by Jochen [Topf 2001], although we independently developed the same technique as part of our investigation on puppetnets. Our work goes one step further by exploring how such techniques can be misused by attackers that control a large number of browsers. A scanning approach that is somewhat similar to how puppets could propagate worms is imagined by Weaver *et al.* in [Weaver et al. 2004], but only in the context of a malicious Web page directing a client to create a large number of requests to nonexistent servers with the purpose of abusing scan blockers. The misuse of JavaScript for attacks such as scanning behind firewalls was independently invented by Grossman and Niedzialkowski [Grossman and Niedzialkowski 2006] while our study was in review [Lam et al. 2005].

The reconnaissance technique relies on the same principle used for timing attacks against browser privacy [Felten and Schneider 2000]. Similar to our probing, this attack relies on timing accesses to a particular website. In our case, we use timing information to infer whether the target site exists or is unreachable. In the case of the Web privacy attack, the information is used to determine if the user recently accessed a page, in which case it can be served instantly from the browser cache.

Puppetnets are malicious distributed systems, much like reflectors and botnets. Reflectors have been analyzed extensively by Paxson [Paxson 2001]. Reflectors are regular servers that, if targeted by appropriately crafted packets, can be misused for DDoS attacks against third parties. The value of reflectors lies both in allowing the attacker to bounce attack packets through a large number of different sources, hereby making it harder for the defender to develop the necessary packet filters, as well as acting as amplifiers, given that a single packet to a reflector can trigger the transmission of multiple packets from the reflector to the victim.

There are several studies discussing botnets. Cooke *et al.* [Cooke et al. 2005] have analyzed IRC-based botnets by inspecting live traffic for botnet commands as well as behavioral patterns. The authors also propose a system for detecting botnets with advanced command and control systems using correlation of alerts. Other studies of botnets include [The Honeynet Project 2005; Li et al. 2005]. From our analysis it becomes evident that botnets are much more powerful than puppetnets and therefore a much larger threat. However, they are currently attracting a lot of attention, and may thus become increasingly hard to setup and manage, as end-point and network-level security measures continue to focus on botnets.

5. CONCLUDING REMARKS

We have explored a new class of Web-based attacks that involve malicious or subverted websites manipulating their visitors towards attacking third parties. We have shown how attackers can set up powerful malicious distributed systems, called Puppetnets, that can be used for distributed DoS, reconnaissance probes, worm propagation and other attacks. We have attempted to quantify the effectiveness of these attacks, demonstrating that the threat of puppetnets is significant. We have also discussed several directions for developing defenses against puppetnet attacks. None of the strategies were completely satisfying, as most of them offered only partial solutions. Nevertheless, if implemented, they are likely to significantly reduce the effectiveness of puppetnets.

Acknowledgments

We thank S. Sidiroglou, S. Ioannidis, M. Polychronakis, E. Athanasopoulos, E. Markatos, M. Greenwald, the members of the Systems and Security Department at I^{2}R and the anonymous reviewers for very insightful comments and suggestions on earlier versions of this work. We also thank Blue Martini Software for the KDD Cup 2000 data.

REFERENCES

2004. Mozilla Port Blocking. `http://www.mozilla.org/projects/netlib/PortBanning.html`.

ABC ELECTRONIC. 2006. ABCE Database. `http://www.abce.org.uk/cgi-bin/gen5?runprog=abce/abce&noc=y`.

ALCORN, W. 2005. The cross-site scripting virus. `http://www.bindshell.net/papers/xssv/xssv.html`. Published: 27th September, 2005. Last Edited: 16th October 2005.

ALEXA INTERNET INC. 2006. Global top 500. `http://www.alexa.com/site/ds/top_500`.

ANDERSEN, S. AND ABELLA, V. 2004. Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 2: Network Protection Technologies. Microsoft TechNet, `http://www.microsoft.com/technet/prodtechnol/winxppro/maintain/sp2netwk.mspx`.

ANONYMOUS. 2004. About the Alexa Toolbar and traffic monitoring service: How accurate is Alexa? `http://www.mediacollege.com/internet/utilities/alexa/`.

BARRETT, B. L. 2005. Home of the webalizer. `http://www.mrunix.net/webalizer`.

BERK, V., BAKOS, G., AND MORRIS., R. 2003. Designing a framework for active worm detection on global networks. In *Proceedings of the IEEE International Workshop on Information Assurance*.

BERNERS-LEE, T., MASINTER, L., AND MCCAHILL, M. 1994. Uniform Resource Locators (URL). *RFC 1738*.

BORTZ, A., BONEH, D., AND NANDY, P. 2007. Exposing private information by timing web applications. In *Proceedings of the* 16^{th} *International World Wide Web Conference*.

CERT. 2000. Advisory CA-2000-02: Malicious HTML Tags Embedded in Client Web Requests. `http://www.cert.org/advisories/CA-2000-02.html`.

CERT. 2001a. Advisory CA-2001-19: 'Code Red' Worm Exploiting Buffer Overflow in IIS Indexing Service DLL. `http://www.cert.org/advisories/CA-2001-19.html`.

CERT. 2001b. Vulnerability Note VU#476267: Standard HTML form implementation contains vulnerability allowing malicious user to access SMTP, NNTP, POP3, and other services via crafted HTML page. `http://www.kb.cert.org/vuls/id/476267`.

CHINCHANI, R. AND BERG, E. V. D. 2005. A fast static analysis approach to detect exploit code inside network flows. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)*.

CHOU, N., LEDESMA, R., TERAGUCHI, Y., AND MITCHELL, J. 2004. Client-side defense against web-based identity theft. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS '04)*.

CLAESSENS, J., PRENEEL, B., AND VANDEWALLE, J. 2002. A tangled world wide web of security issues. *First Monday 7*, 3 (March).

COOKE, E., JAHANIAN, F., AND MCPHERSON, D. 2005. The Zombie Roundup: Understanding, Detecting, and Disrupting Botnets. In *Proceedings of the 1st USENIX Workshop on Steps to Reducing Unwanted Traffic on the Internet (SRUTI 2005)*.

FELTEN, E. W., BALFANZ, D., DEAN, D., AND WALLACH, D. S. 1997. Web Spoofing: An Internet Con Game. In *Proceedings of the 20th National Information Systems Security Conference*. 95–103.

FELTEN, E. W. AND SCHNEIDER, M. A. 2000. Timing attacks on Web privacy. In *Proceedings of the 7th ACM Conference on Computer and Communications Security (CCS'00)*. ACM Press, New York, NY, USA, 25–32.

GARRETT, J. J. 2005. Ajax: A New Approach to Web Applications. `http://www.adaptivepath.com/publications/essays/archi-ves/000385.php`.

GLADYCHEV, P., PATEL, A., AND O'MAHONY, D. 1998. Cracking RC5 with Java applets. *Concurrency: Practice and Experience 10*, 11-13, 1165–1171.

GROSSMAN, J. AND NIEDZIALKOWSKI, T. 2006. Hacking intranet websites from the outside - javascript malware just got a lot more dangerous. Blackhat USA.

HEALAN, M. 2003. Referer spam. `http://www.spywareinfo.com/articles/referer_spam/`.

INC., W. 2006. Webtrends web analytics and web statistics. `http://www.webtrends.com`.

IOANNIDIS, S. AND BELLOVIN, S. M. 2001. Building a Secure Browser. In *Proceedings of the Annual USENIX Technical Conference, Freenix Track*.

JACKSON, C., BORTZ, A., BONEH, D., AND MITCHELL, J. C. 2006. Protecting browser state from Web privacy attacks. In *Proceedings of the WWW Conference*.

JIM, T., SWAMY, N., AND HICKS, M. 2007. Defeating scripting attacks with browser-enforced embedded policies. In *Proceedings of the 16th International World Wide Web Conference (WWW'07)*.

KEIZER, G. 2005. Dutch botnet bigger than expected. `http://informationweek.com/story/showArticle.jhtml?articleID=172303265`.

KEPHART, J. O. AND WHITE, S. R. 1991. Directed-graph epidemiological models of computer viruses. In *Proceedings of the 1991 IEEE Computer Society Symposium on Research in Security and Privacy*.

KOHAVI, R., BRODLEY, C., FRASCA, B., MASON, L., AND ZHENG, Z. 2000. KDD-Cup 2000 organizers' report: Peeling the onion. *SIGKDD Explorations 2*, 2, 86–98.

KORPELA, E., WERTHIMER, D., ANDERSON, D., COBB, J., AND LEBOFSKY, M. 2001. SETI@home – Massively Distributed Computing for SETI. *Computing in Science & Enginering 3*, 1, 78–83.

KRUEGEL, C., KIRDA, E., MUTZ, D., ROBERTSON, W., AND VIGNA, G. 2005. Polymorphic worm detection using structural information of executables. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)*.

KRUEGEL, C. AND VIGNA, G. 2003. Anomaly detection of Web-based attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS'03)*. ACM Press, New York, NY, USA, 251–261.

LAM, V. T., ANTONATOS, S., AKRITIDIS, P., AND ANAGNOSTAKIS, K. G. 2005. PuppetNet project website. http://s3g.i2r.a-star.edu.sg/proj/puppetnets.

LAM, V. T., ANTONATOS, S., AKRITIDIS, P., AND ANAGNOSTAKIS, K. G. 2006. Puppetnets: Misusing web browsers as a distributed attack infrastructure (extended version). Technical Report, http://s3g.i2r.a-star.edu.sg/proj/puppetnets.

LI, J., EHRENKRANZ, T., KUENNING, G., AND REIHER, P. 2005. Simulation and analysis on the resiliency and efficiency of malnets. In *Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation (PADS'05)*. IEEE Computer Society, Washington, DC, USA, 262–269.

LITTLE, J. D. C. 1961. A Proof of the Queueing Formula $L = \lambda W$. *Operations Research* 9, 383–387.

MAONE, G. 2006. Firefox add-ons: Noscript. `https://addons.mozilla.org/firefox/722/`.

MONIZ, D. AND MOORE, H. 2006. Six degrees of xssploitation. Blackhat USA.

MOZILLA.ORG. 2004. End User Guide: Automatic Proxy Configuration (PAC). `http://www.mozilla.org/catalog/end-user/customizing/enduserPAC.html`.

NACHENBERG, C. 1997. Computer virus-antivirus coevolution. *Commun. ACM 40,* 1, 46–51.

PAXSON, V. 2001. An analysis of using reflectors for distributed denial-of-service attacks. *ACM Computer Communication Review 31,* 3, 38–47.

PHILIPPINE HONEYNET PROJECT. Philippine Internet Security Monitor - First Quarter of 2006. `http://www.philippinehoneynet.org/docs/PISM20061Q.pdf`.

POLYCHRONAKIS, M., ANAGNOSTAKIS, K. G., AND MARKATOS, E. P. 2006. Network-level polymorphic shellcode detection using emulation. In *Proceedings of the GI/IEEE SIG SIDAR Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA).*

REIS, C., DUNAGAN, J., WANG, H. J., DUBROVSKY, O., AND ESMEIR, S. 2006. Browsershield: Vulnerability-driven filtering of dynamic html. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*.

RIZZO, L. 1997. Dummynet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review 27,* 1, 31–41.

ROSS, B., JACKSON, C., MIYAKE, N., BONEH, D., AND MITCHELL, J. C. 2005. Stronger password authentication using browser extensions. In *Proceedings of the 14th Usenix Security Symposium*.

RUBIN, A. D. AND JR., D. E. G. 1998. A Survey of Web Security. *IEEE Computer 31,* 9, 34–41.

RUDERMAN, J. 2001. The Same Origin Policy. `http://www.mozilla.org/projects/security/components/same-origin.html`.

SAROIU, S., GUMMADI, P., AND GRIBBLE, S. 2002. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking (MMCN).*

SCHNEIER, B. 2005. Attack trends 2004 and 2005. *ACM Queue 3,* 5 (June).

SMITH, F., AIKAT, J., KAPUR, J., AND JEFFAY, K. 2003. Variability in TCP round-trip times. In *Proceedings of the 3rd ACM SIGCOMM Conference on Internet measurement*.

STAMM, S., RAMZAN, Z., AND JAKOBSON, M. 2006. Drive-by pharming. In *Technical Report TR641, Indiana University Department of Computer Science*.

STANIFORD, S., MOORE, D., PAXSON, V., AND WEAVER, N. 2004. The top speed of flash worms. In *Proc. ACM WORM*.

STANIFORD, S., PAXSON, V., AND WEAVER, N. 2002. How to Own the Internet in Your Spare Time. In *Proceedings of the* 11^{th} *USENIX Security Symposium*. 149–167.

STUNNIX. 2006. Stunnix javascript obfuscator - obfuscate javascript source code. `http://www.stunnix.com/prod/jo/overview.shtml`.

SYMANTEC. 2005. Internet Threat Report: Trends for January 05-June 05. Volume VIII. Available from www.symantec.com.

TECHWEB.COM. 2004. Lycos strikes back at spammers with dos screensaver. `http://www.techweb.com/wire/security/54201269`.

THE HONEYNET PROJECT. 2005. Know your enemy: Tracking botnets. `http://www.honeynet.org/papers/bots/`.

TOPF, J. 2001. HTML Form Protocol Attack. `http://www.remote.org/jochen/sec/hfpa/`.

VNEXPRESS. 2005. Website of largest Vietnamese hacker group attacked by DDoS. `http://vnexpress.net/Vietnam/Vi-tinh/2005/12/3B9E4A6D/`.

VOGT, P., NENTWICH, F., JOVANOVIC, N., KIRDA, E., KRUEGEL, C., AND VIGNA, G. 2007. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proceedings of the 13th Annual Network and Distributed System Security Symposium (NDSS '07).*

WANG, D. 2005. HOWTO: ISAPI Filter which rejects requests from SF_NOTIFY_PREPROC_HEADERS based on HTTP Referer. `http://blogs.msdn.com/david.wang`.

WANG, Y. AND WANG, C. 2003. Modeling timing parameters for virus propagation on the internet. In *Proceeding of the 1st Workshop Of Rapid Malcode (WORM'03).*

WANG, Y.-M., BECK, D., JIANG, X., ROUSSEV, R., VERBOWSKI, C., CHEN, S., AND KIN, S. 2006. Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites That Exploit Browser Vulnerabilities. In *Proceedings of the 13th Annual Network and Distributed System Security Symposium (NDSS '06)*.

WEAVER, N., STANIFORD, S., AND PAXSON, V. 2004. Very Fast Containment of Scanning Worms. In *Proceedings of the* 13th *USENIX Security Symposium*. 29–44.

WILLIAMS, A. T. AND HEISER, J. 2004. Protect your PCs and Servers From the Bothet Threat. Gartner Research, ID Number: G00124737.

ZONE-H. 2006. Digital attacks archive. `http://www.zone-h.org/en/defacements/`.

ZOU, C. C., GONG, W., AND TOWSLEY, D. 2002. Code Red Worm Propagation Modeling and Analysis. In *Proceedings of the* 9th *ACM Conference on Computer and Communications Security (CCS)*. 138–147.

...