The Little Hybrid Web Worm that Could

Billy Hoffman
Lead Researcher
SPI Dynamics
(bhoffman@spidynamics.com)

John Terrill
Executive Vice President and co-founder
Enterprise Management Technology
(jterrill@http://www.em-technology.net/)

Abstract

The past year has seen several web worm attacks against various online applications. While these worms have gotten more sophisticated and made use of additional technologies like Flash and media formats, they all have had some basic limitations such as infecting new domains and injection methods. These worms are fairly easily detected using signatures and these limitations have made web worms annoying, but ultimately controllable. This paper examines the possibility of *hybrid web worms* which use several methods to overcome the limitations of current web worms. Specifically the authors examine how a hybrid web worm: mutates itself to evade defenses; updates itself with new attack vectors while in the wild; and finds and exploits targets regardless of whether they are client web browsers or web servers.

Current Web Worms

Web worms are a form of self propagating malware that exploit web applications. The past year has seen several web worm attacks against various online applications (1). While these worms have gotten more sophisticated and make use of additional technologies like Flash and media formats, current web worms have numerous flaws. They have limited abilities to infect other hosts. Many worms, such as Samy, can only infect a single host (2). Those that can infect other hosts choose these hosts in a poor or predictable manner. For example, Perl.Santy used a static string to query Google (3). Google could detect search queries made by infected hosts and stopped the worm from propagating by returning an error page instead of search results. Current web worms have counter measures to defeat security products or evade signatures. Many anti-virus products have signatures for known web worms. Current web worms are vulnerable to single “silver bullet” fixes. Those that are confined to a single host die when that website fixed the underline security vulnerability and flushed their database of the malicious content. Those that are cross domain only exploit 1 or 2 vulnerabilities (3).The overall survivability of current web worms is quite poor.

Hybrid Web Worms

Hybrid web worms attempt to overcome many of these limitations. A hybrid web worm is a worm that can run on both a web browser or a web server. This allows the hybrid web worm to take advantage of the enormous number of Cross-site Scripting vulnerabilities, but while allowing the same worm to utilize the rarer but more powerful command execution vulnerability on a web server (4). These two different execution models are stored in the same worm and this allows hybrid web worm to survive various situations. Consider Figure 1 which shows a hybrid web worm injecting both server-side web servers and client-side web browsers.

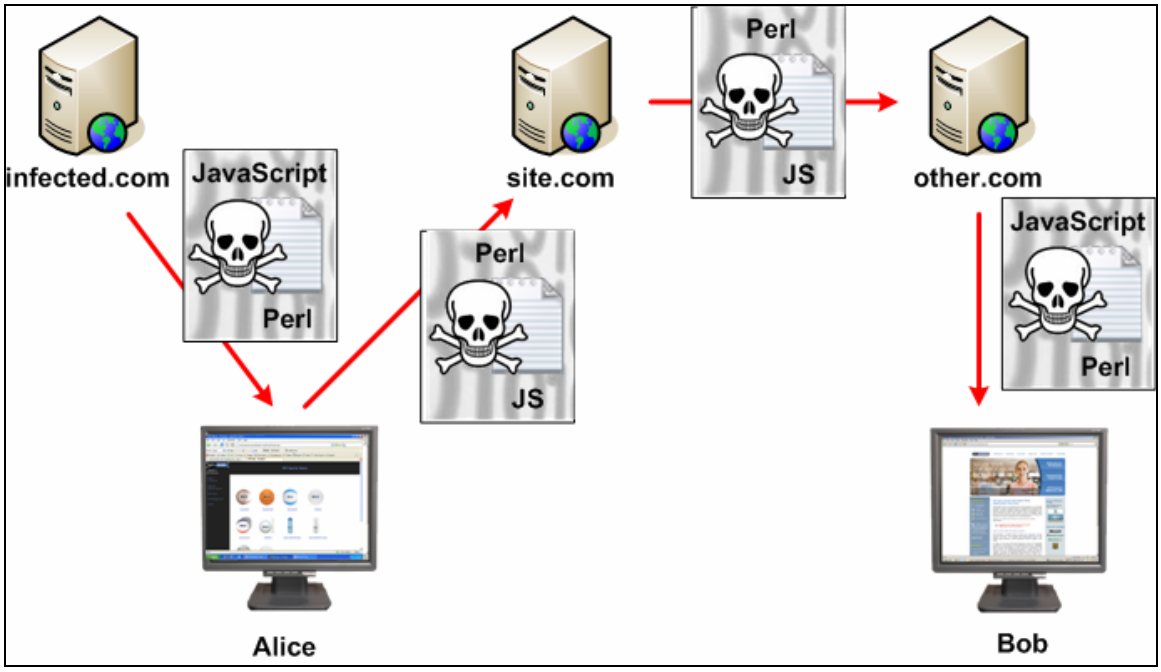

Figure 1, A hybrid web worm as it infects both web servers and web clients.

Alice visits `infected.com` and receives the JavaScript version of the hybrid web worm. The Perl version of the worm lies dormant inside the JavaScript version. The JavaScript code runs in Alice's browser and uses Alice's machine to exploit a command execution vulnerability on `site.com`. The JavaScript version of the worm injects the Perl version of the worm (with a copy of the JavaScript version inside) into `site.com`. On the web server hosting `site.com` the Perl code runs and injects the Perl version into another vulnerable web server, this time located at `other.com`. When the code runs on the web server hosting `other.com` the hybrid web worm writes the JavaScript version of itself into the web pages on `other.com`. When Bob visits `other.com` the JavaScript version of the hybrid web worm is downloaded to his browser and the process continues.

Hybrid worms are designed to work in as many situations as possible. They are written in interpreted languages for maximum portability. For the client-side portion this means the worm is written in JavaScript or possibly VBScript or ActionScript in a Flash object. Perl, PHP, Python or Ruby are all possible server-side languages.

In addition to running in multiple environments, hybrid web worms also mutate their source code as they propagate to evade security systems or anti-virus programs. Hybrid worms can also

update themselves with new vulnerabilities to exploit while in the wild. All of these features are designed to increase the lifespan of the worm as discussed at Black Hat Federal 2006 (5).

A hybrid worm typically has more capabilities when running on a web server than when running on the client in a web browser. For example, it may have access to native commands on the web server such as *netcat* or *wget* allowing it to easily conduct HTTP transactions with arbitrary sites. It might also have access to full language interpreters such as Perl, PHP, Python or Ruby. In short when running on the server the hybrid worm is only limited by the capabilities of the web server's user id. When running on the client the hybrid worm is restricted by the limits the browser places on JavaScript. These limitations are typically greater than the limits of the web server. Thus this paper spends more time discussing on how the client-side portion of a hybrid worm could conduct various actions and only briefly discusses how such actions would be executed on the web server.

This paper will examine how a hybrid web worm performs the following:

1. Evade detection from signature based security systems such as anti-virus or IDS.
2. Adapt new attack vectors while in the wild to prevent "silver bullet" fixes from halting infection.
3. Find and infect new hosts running on multiple domains.
4. Various payloads available to the worm.

Current Malware Detection

There are orders of magnitude more benign JavaScript programs than malicious JavaScript. As a result many security vendors are creating blacklists of known malicious JavaScript. For example, Symantec's anti-virus products detect the Yamanner worm using blacklist signatures (6).

As a result some attackers are starting to obfuscate their code. This is mainly an innovation used by attackers leveraging obfuscated JavaScript to bootstrap traditional attacks against the browser or operating system (7) (8). Web worms have yet to use these obfuscation techniques to any great extent beyond getting the initial worm injected into a site. Early obfuscation methodologies such as JS/Wonka used the same decoding function with different decoding data (9). Companies like Symantec responded by creating signatures for the common decoding function. More advanced obfuscation techniques using so-call "dynamic obfuscation" mirrors the typical approach of polymorphic virus engines of the early 1990s such as DAME (8) (10). A block of encrypted program code and a decode function are mutated with each generation. Security firms like Websense and Finjan apparently have proprietary methods for detecting these advanced methods (7) (8). The authors discovered their own detection method. The ratio of bytes devoted to literals in relation to the total size of the program is very high (over 30%) in encrypted block malware. Normal JavaScript has a ratio of characters in literals to total characters in the program of between 2%-7%.

Evading Detection

Hybrid web worms attempt to evade detection from security products by mutating the source code to prevent all mutations of the code from containing a common signature. The authors chose not to decrypt and dynamically run malicious code as discussed above because (as mentioned above) the authors discovered a way to defeat that method. Instead the source code for the hybrid worm is directly visible and this source code and all literals are mutated with each infection. While there has been public discussion about mutating interpreted web languages before (5) (11) the authors could not find any published proof of concepts.

For this paper, we consider two types of source code mutation. A *reversible mutation* is a mutation which produces code that can be further mutated using some mutation function. A *final mutation* is a mutation which produces code that cannot be mutated further using some mutation function. A security product could potentially create a signature for this final "steady-state" to detect malicious code. To prevent a final "steady state" of the code that cannot be further mutated it is paramount to minimize the number of *final mutation* algorithms in the hybrid worm. Assuming mutations are chosen with the same probability, the probability that the N^{th} generation of code can be mutated further is given by the formula:

$$\left(\frac{\text{Number of Reversible Mutations}}{\text{Total Number of Mutations}} \right)^{N}$$

For example, if you had five mutation functions, four which were reversible and one which was final, then after 14 generations a sample of code has only a 4% chance of mutating further.

Another concern when mutating code is increasing the size of successive generations. If mutations only increase the length of the malware will grow without size, inhibiting its ability to effectively transport itself from host to host. There must be a corresponding mutation that can undo these size increases to prevent the source from growing without bounds. In general mutations should be throttled to ensure too much code does not change too quickly to ensure maximum diversity.

The authors implemented their mutation engine using regular expressions to locate various code structures and a list of literals. This prevents the mutation engine from matching on what appears to be a control structures but is really just part of a string literal. The performance was quite good though the lack of a complete tokenizer and parser limited the full range of mutations that were possible.

Mutating Control Flow Structures

Many control flow structures can be mutated into other forms of control structures as the following table shows. Each row represents a 1-way operation from *Code Structure 1* to *Code Structure 2*. Notice that the all mutations are reversible (everything in *Code Structure 2* matches a format in *Code Structure 1*) which prevents the source code from reaching a steady state.

Code Structure 1	*Code Structure 2*

`do {`
`  code;`
`} while(conditional);`
	`code;`
	`while(conditional) {`
	`  code;`
	`}`

`while(conditional) {`	`for(;conditional;) {`

code; }	code; }

for(init; conditional; step) {
code;
}
	init;
	while(conditional) {
	code;
	step;
	}

if(conditional) {
code;
}
	while(conditional) {
	code;
	break;
	}

if(conditional) {
code1;
} else (
code2;
}
	if(!(conditional)) {
	code2;
	} else {
	code1;
	}

if(conditional) {
code1;
} else (
code2;
}
	var tmp = false;
	while(conditional) {
	code1;
	break;
	tmp = true;
	}
	while(!tmp) {
	code2;
	break;
	}

if(variable == literal1) {
code1;
} else if(variable == literal2) {
code2;
} else {
code3;
}
	switch(variable) {
	case literal1:
	code1;
	break;
	case literal2:
	code2;
	break;
	default:
	code3;
	}

Also note that some code structures are less likely to survive. There is not a mutation to convert a code block into a `do…while` loop. This means any `do…while` loops in the original source code will quickly be mutated away.

Literal Expansion and Collapsing

The mutation engine will expand and collapse string or numeric literals. For example, a string literal such as `"spi dynamics"` can be broken into pieces and concatenated together such as `"spi dy"+"namics"` or `"spi dynami"+"ics"`. It should be noted that each time a string literal is expanded using this method overall length of the worm increases by 3 characters for the additional `"+"`. Another option is to convert a string literal into a sequence of character codes

such as `String.fromCharCode(115,112,105,32,100,121,110,97,109,105,99,115)`. This is beneficial because it converts string literals into numeric literals which can be mutated into more forms than string literals. A major downside is this method significantly increases to the size of the mutated code (the `String.fromCharCode` notation is over 500% of the original string literal).

Numeric Literals offer many opportunity for mutation. First of all literals can be represented using various number bases such as decimal, octal, or hexadecimal. Thus the 115 in decimal can be written as 0163 in octal (octal literals have a leading 0 in JavaScript) or 0x73 in hexadecimal. Numeric literals can be expanded much like string literals can. Generally, a numeric literal can be expanded by performing two mathematical operations which cancel each other out. These take the form (((x) op1 y) op2 y) where x is the original numeric literal, y is a randomly generated numeric literal and op1 and op2 are 2 inversely related operations. The following table shows the pairs of operators that should be used together.

Operation 1	*Operation 2*
Addition	Subtraction
Subtraction	Addition
Multiplication	Division
Division	Multiplication

Care must be taken to also write a collapsing code function for every literal expansion function. Collapsing code that is code that is capable of detecting the literal expansions and collapsing them back into the original form. For example, the mutation engine should be ab;e to collapse the four parts of `"s"+"pi dyna"+"mi"+"ics"` into 3 parts such as `"s"+"pi dyna"+"mics"` (which can then be expanded into `"s"+String.fromCharCode(112,105,32,100,121,110,97)+"mics"`). Collapsing code prevents our mutations from growing the code without bounds.

Variable Renaming

Another obvious mutation is to change all the variable and function names. However, unless the word generating algorithm is designed with care detectable artifacts can be introduced into the hybrid web worm. A common (and misguided idea) is to simply randomly select letters. This produces completely gibberish variables names such as qhgzl or vogpmr. While source code is by no means English literature, variables names are often composed of English words or English abbreviations. Simple cryptographic analysis could be used to find variables that were randomly generated without any thought to letter selection. For example, on average English words are 5.1 letters in length (12) and average of 31% - 37% of the letters are vowels[1]. Basic letter and digraph frequency also enabled the creation of more English-like “words” (13). The author’s source code mutation engine incorporated these characteristics of the English language to generate random camel case variable names such as *thonManny* that look like English words but are not.

Another viable method (which was not used by the authors) involves using the words on the web page the hybrid worm is injected into. Consider the following block of JavaScript:

[1] The authors ran tests against multiple public domain English tests from Project Gutenberg including *Ulysses* by James Joyce, *Alice's Adventures in Wonderland* by Lewis Carroll, and *Pride and Prejudice* by Jane Austen. While this might induce a late 1800s British skew into the results, the authors spot checked the 31%-37% range against *Rainbow 6* by Tom Clancy and *Jurassic Park* by Michael Crichton and similar results.

```
function extractWords(txt) {
    var words = {};
    var tmp = txt.toLowerCase().split(' ');
    for(var i=0; i< tmp.length; i++) {
        //strip leading and trailing white space
        tmp[i] = tmp[i].replace( /^\s+/g, "" );
        tmp[i] = tmp[i]..replace( /\s+$/g, "" );
        if(tmp.length) {
            //if there is anything there, add it
            //hash table avoid dups
            words[tmp[i]] = true;
        }
    }
    return wordsArray;
}
var useableWords = extractWords(document.body.innerHTML);
```

This code will extract all the words in the body of a web page and present a list without duplicates. These words provide a good resource to aid variable renaming. A more effective solution might be walking the DOM tree and only extract words from text nodes to ensure only English text is included in the resulting list.

Inserting Non-code Elements

The authors explored but did not implemented inserting random whitespace or comments to evade signature-based detection mechanisms. Inserting whitespace does not provide much benefit because it is trivial to create signatures (especially regular expression based signatures) that ignore whitespace. Both extra random whitespace and comments represent non-code elements. A language tokenizer immediately discards these elements. The authors believe that deeper code inspection methods using language tokenization and parsing represent the next step security vendors will need to take in attempting to detect malicious code. As such, inserting more non-code elements provides no protection from these new methods. Non-code elements actually hurt the hybrid worm because they increase the size of the worm.

Other Possible Mutations

There are other mutations which may seem obvious to the reader that the authors did not implement in their source code mutation engine. Typically other mutations were not included because of the difficultly in reversing the mutation. For example, adding do-nothing code such as `if(false) {…}`, `(x OR 0)`, or `(x AND x)` and detecting/removing it without potentially altering the original code functionality is difficult without complex language tokenizing and parsing code. Mutating branch structures is also difficult without complex language tokenizing and parsing code. The simple logic in the authors' source code mutation engine is the prime reason for only a few branch structure mutations.

Mutation is not just limited to interchangeable logic structures but also spans to communication functions. For example, the hybrid web worm might use an `Image` object to send data to back to an attacker. This can be mutated to using an `OBJECT` tag or `FORM` tag in later generations.

Updating Attack Vectors

All worms have a pool of potentially exploitable systems. This pool is defined by the number of hosts vulnerable to a given vulnerability, the ease of discovering those hosts, and whether those hosts are reachable from infected hosts. Once a worm is released, the size of this pool shrinks as people take defensive measures such as patching or removing their systems from public networks. And while it is possible for this pool to expand after the worm has been released (a patched machine is re-imaged with an unpatched image, new, unpatched machines are placed online, etc) typically this is not the case.

Worms which exploit a single vulnerability have a smaller pool of potentially infectable machines than worms which exploit multiple vulnerabilities. By leveraging multiple vulnerabilities the hybrid web worm has a better chance of locating vulnerable hosts. However, even worms that exploit multiple vulnerabilities will eventually run out of vulnerable targets that are discoverable and reachable. Only by adding new attack vectors while the worm is in the wild can the worm significantly grow its pool of potentially exploitable systems.

There are two ways the hybrid web worm can learn about new attack vectors while in the wild: by retrieving information on known vulnerabilities from a public website or by independently discovering the unknown vulnerabilities themselves.

Fetching New, Known Vulnerabilities

Many neutral (i.e. non-attacker controlled) websites publish information about new web application vulnerabilities. Obviously this occurs on websites that archive mailing lists like Full Disclosure. However advisories on mailing lists vary greatly in terms of quality and do not have a standardized format. Furthermore these vulnerabilities have not been confirmed as real. Security sites like Secunia provide a clearing house for web vulnerabilities. Advisories published on `secuncia.com` have been verified and the advisories have a standard, repeatable structure that allows simple regular expressions to extract out information about the vulnerability.

For example, Secunia vulnerabilities that contain a single issue typically have three or four paragraphs. The first paragraph tells what web component is affected, who discovered it, and what type of vulnerability it is. This paragraph is typically of the format "*[WHO] has reported (a|some) vulnerabilit(y|ies) in [FILE], which can be exploited … to conduct [VULNERABILITY TYPE] attacks."* The second paragraph details which parameters are vulnerable and contains a phrase such as "*Input passed to the [VULNERABLE PARAMETER] parameter…*" The final paragraph contains information about which versions are vulnerable. If only a specific version is vulnerable the paragraph is of the format "*The vulnerabilit(y|ies) (is|are) reported in version [VERSION]*." If a range of versions are vulnerable the format is "*The vulnerabilit(y|ies) (is|are) reported in versions [OLDEST VERSION] to [NEWEST VERSION]*." There are slight variations with regards to plurals nouns and when multiple parameters or multiple vulnerabilities are disclosed in a single advisory. Also there is an occasional appearance by a paragraphs describing special configuring information required for the vulnerability to function. All of these variations follow common formats and can be handled/mitigated with well written regular expressions.

Another potential source of vulnerabilities are defacer "score board" style sites such as Zone-H or `xssed.com`. Unlike Secunia, these sites list specific websites that are vulnerable and the attack string used to exploit them. This is a much more explicit description of the attack vector allowing the hybrid web worm to know exactly where to insert its attack payload. `xssed.com` is an

especially good resource for up-to-date Cross-site Scripting vulnerabilities in a machine consumable format. However, these sites showcase attacks against specific websites as opposed to attacks against specific reusable web components. While this allows the hybrid web worm to exploit specific sites it is less helpful for the long term survivability of the worm than attack vectors disclosed for against a common component present on multiple sites.

Another source of attack vectors would be for an attacker to manually publish machine consumable vulnerability information on multiple public and highly mirrored mailing lists. This provides a best of both worlds scenario in the attacker can supply the hybrid web worm with new and very specific attack vector information without needing a single bottleneck website they control that can be blocked.

Discovering New Vulnerabilities

The hybrid worm could also attempt to find new vulnerabilities on its own using a web vulnerability scanner. While on the server, the hybrid might be able to use *nmap* to find new targets on the web server's intranet and use *Nikto* (14) to find vulnerabilities to inject itself into. On the client the web vulnerability scanner *Jikto* could be used (15). Activities like port scanning and vulnerability scanning can take large amounts of time, especially when done inside of an interpreted program running inside of a browser which has HTTP connection limitations. Offline Ajax frameworks such as Google Gears (16) provide a threading model to allow large JavaScript jobs to run without interruption. This could make client-side scanning applicable in more situations. The authors have conducted further research in the area.

Finding and Infecting New Hosts

The authors examined three different methods a hybrid web worm can use to discover new hosts to infect while in the wild: port scanning for new targets, retrieving a list of new targets from a controlling 3rd party (ala XSS-Proxy (17) or Backframe (18)), and querying search engines for new targets. This section focuses on using search engines to locate targets.

Due to JavaScript's Same Origin Policy, it is difficult for a hybrid worm running on the client to query a search engine for new targets and be able to read the response. Over the last year numerous methods have been discovered and refined such as IE's MHTML (19), bypassing the Same Origin Policy using websites that "proxy" a website into their security domain (20), and using Web APIs (21). The authors use the cross domain communication method using "proxy" websites first discovered by Petko Petkov (20) and further refined by one of the authors for Jikto (15) because we are most familiar with it. This method has the added benefits of working cross-browser and cross platform (a must for a worm), doesn't require special circumstances, and doesn't rely on a single mashup or API that could change without notice. Websites that provide "proxy" services are everywhere. However the method of cross browser communication is not important and any method can be used depending on the situation. Figure 2 illustrates the typical use of cross domain communication using "proxy" websites.

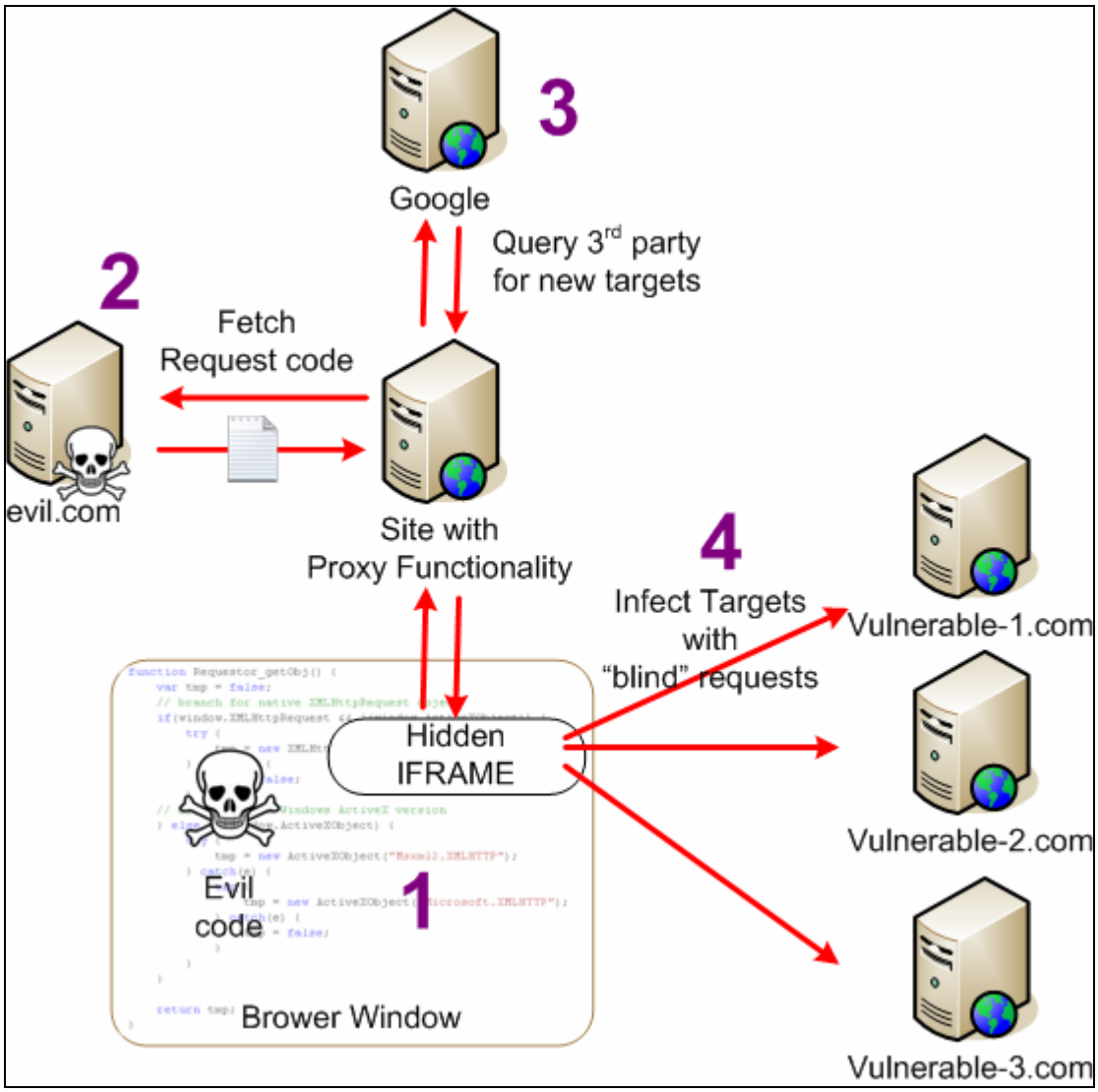

Figure 2: A hybrid web worm using a site with proxy functionality to find and attack new targets.

The first step the hybrid performs is creating an `IFRAME` pointed at a site that provides proxy functionality. In the next step, the worm uses this `IFRAME` to download JavaScript from `evil.com` into the security domain of the proxy site. This allows the JavaScript to side-step the Same Origin Policy and use Ajax to contact the proxy site and send search requests to Google to find possible targets. Once these targets are located, the hybrid web worm can send blind `GET`s and `POST`s to the target websites infecting them with the new mutated copies of the hybrid web worm.

When the hybrid worm runs on the server, the worm simply queries a search engine like Google directly for targets.

How a web worm queries a search engine for targets is very important. So-called Google hacking provides a means for using search engines to find target websites that are running vulnerable versions of specific web components (22). A query for *"Powered By XYZ version 4.1"* is not enough. The Perl.Santy worm used Google to find websites running a vulnerable versions of phpBB (5). However the search query it used to locate these targets was static. Google was able

to stop the worm from spreading by simply returning an error page whenever it received a query with Perl.Santy's static search string. This is another example of worms or worm activity being detected and stopped by creating signatures for static behavior of the worm. The hybrid web worm creates dynamic search strings to prevent the search engine from detecting its queries looking for new targets. First of all the hybrid worms adapt to use new attack vectors and vulnerabilities so the fundamental content of its search string (the content to find particular versions of web components) will naturally change on its own. In addition, the hybrid worm adds a random number of random words to the query. For example the query may look like *"Powered By phpBB" -window -cats* which will find vulnerable sites that don't contain the words *window* or *cats*. Another option is to use completely made up words. The search string *"Powered By phpBB" OR nvbsgetalk* returns essentially the same results as "Powered By phpBB" because ORing in the results for a nonexistent word does not really change things. Both algorithms for generating real English words and English-like words from our *Variable Renaming* section above can provide words to mutate a search query.

Once the worm has a list of new targets it attempts to infect them. The context of the hybrid worm dictates the methods it can use to spread to other targets. When running on the client, the worm can use various methods such as the Image object, various HTML tags, and and IFrames to send blind HTTP GETs and POSTs to other domains (23). Using a CSS and JavaScript the worm could determine which sites a user has visited or which sites they are logged into thus having a higher probability of leveraging cached login credentials to propagate(24)(25). There are large array of infection options available when the hybrid web worm is running on the server. With a command execution vulnerabilities attacks can load and execute full programs of their choice. Other options are available with PHP or Perl injection vulnerabilities. Common methods would include using fopen, Perl:LWP, Sockets, or any other available network or file based functions. As mentioned earlier, the server-side code can write the client-side version of itself into the webpages hosted on the web server. This way the client-side version of the hybrid worm is sent to new prospective victims that are browsing websites hosted on the web server.

Worm Payloads

The payloads of web worms can vary and depend on where the worm is executing. When executing on the client, all the nasty JavaScript techniques discovered in recent years are applicable, including session hijacking, port scanning, keystroke and mouse movement logging, theft of content, website history and search engine query theft. More advanced web attacks are possible, such as Jikto, a web security scanner written in JavaScript or DOMinatrix. DOMinatrix is a SQL Injection tool written by the authors entirely in JavaScript and built upon the techniques of Jikto. Only verbose SQL injection against MSSQL is supported, though it is trivial to add more. Client-side code could also serve to bootstrap traditional attacks that exploit flaws in the browser or file parsing libraries. For example, flaws such as the WMF, VML or MDAC vulnerabilities could be used to load traditional malware onto a user's machine.

When executing on the server, more options are available. When exploiting a command Execution vulnerability the hybrid worm can launch tools or commands on the target as the user id of the service infected. It is further possible for the payload to leverage a local exploit to escalate privileges for administrator access or increased ability. This can also allow for the insertion of kernel level backdoors to ensure the worms duration on the target. Another possibility is creating malicious data files or programs on the server in hope that a user more privileged than the web server interacts with them.

Defense

While researching this topic, the authors came across two potential ways to detect a hybrid web worm.

One possible detection method is to examine the Cyclomatic Complexity or McCabe Complexity of a piece of arbitrary JavaScript code (26). The overall complexity diagram and number of closed loops should remain almost identical regardless of the number of mutations performed on the code. This follows since our mutations change the syntax of the code but not the underline functionality then the complexity of that functionality should remain the same. The authors are investigating whether a complexity diagram alone is capable of uniquely identifying web malware.

Another possible detection method is based on the network traffic signature a hybrid web worm generates. Many of the tasks the worm performs, especially when refining newly discovered attack vectors or when attempting to find new hosts through port scanning or new vulnerabilities through web security scans generate an enormous amount of repeatable traffic. While a hybrid worm's source code mutation may allow it to evade security defenses around a vulnerable host, it could be very easy to identify a host once it has been infected. The authors are investigating how to detect hybrid web worms by their post-infection network traffic.

Regardless of whether it is detectable or not, the hybrid worm functions ultimately by exploiting web application vulnerabilities such as cross-side scripting or command execution. Implementing proper input validation using whitelists that validate both data type, format, and range is a well documented topic (27) and is not repeated in detail here. The authors urge developers to implement proper input validation inside their application and not rely solely on security products such as Firewalls, IDS, IPS, or WAF to provide a wall around a fundamentally broken application.

References

1. *Researchers Warn of Web Worms*, Robert Lemos/Security Focus, http://www.securityfocus.com/news/11405
2. *MySpace Worm Explanation*, Samy, http://namb.la/popular/tech.html
3. *Perl.Santy Information*, Symantec, http://www.symantec.com/security_response/writeup.jsp?docid=2004-122109-4444-99
4. *Web Application Security Statistics*, Web Application Security Consortium, http://www.webappsec.org/projects/statistics/
5. *Analysis of Web Application Worms and Viruses*, Billy Hoffman, http://media.blackhat.com/presentations/bh-federal-06/BH-Fed-06-Hoffman/BH-Fed-06-Hoffman-up.pdf
6. *Malicious Yahooligans*, Symantec, http://www.symantec.com/avcenter/reference/malicious.yahooligans.pdf

7. *Security Labs Alert*, Websense, http://www.websense.com/securitylabs/alerts/alert.php?AlertID=733

8. *Web Security Trends Report Q4 2006*, Finjan Malicious Code Research Center

9. *Analysis of JS/Wonka*, Websense, http://www.websense.com/securitylabs/resource/pdf/wslabs_wonka_analysis_oct05.pdf

10. *Dark Angel's Multiple Encryptor*, 40hex Issue 11, http://www.textfiles.com/magazines/40HEX/40hex011

11. *Doom Or VoMM?*, LMH, http://blog.info-pull.com/2007/01/update-17th-october-2006-aviv-posted.html

12. *Average Word Length of Languages*, Blogamundo, http://blogamundo.net/lab/wordlengths/

13. *English letter and digraph frequency*, http://www.letterfrequency.org/

14. *Nikto*, Cirt.net, http://www.cirt.net/code/nikto.shtml

15. *JavaScript Malware for a Grey Goo Tomorrow*, Billy Hoffman, http://www.spidynamics.com/spilabs/education/presentations/Javascript_malware.pdf

16. *Google Gears*, Google, http://gears.google.com/

17. *XSS-Proxy: A Tool for Realtime XSS Hijacking and Control*, Anton Rager, http://xss-proxy.sourceforge.net/

18. *Backframe*, Petko Petkov, http://www.gnucitizen.org/applications/backframe

19. *IE6.0 and IE7.0 Vulnerable to Complete Cross Domain Leakage*, RSnake, http://ha.ckers.org/blog/20061019/ie60-and-ie70-vulnerable-to-complete-cross-domain-leakage/

20. *Transversing the Web*, Petko Petkov, http://www.gnucitizen.org/blog/traversing-the-web

21. *YPipes Spider*, Petko Petkov, http://www.gnucitizen.org/projects/6th-owasp-conference/spider.htm

22. *Google Hacking*, Johnny Long, http://johnny.ihackstuff.com/

23. *JavaScript Remoting Dangers*, Billy Hoffman, http://www.gnucitizen.org/blog/javascript-remoting-dangers

24. *Hacking Intranet Websites from the Outside*, Jeremiah Grossman, http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Grossman.pdf

25. *I know if You're Logged in Anywhere*, Jeremiah Grossman, http://jeremiahgrossman.blogspot.com/2006/12/i-know-if-youre-logged-in-anywhere.html

26. *A Complexity Measure*, Thomas McCabe, http://www.literateprogramming.com/mccabe.pdf

27. *Data Validation*, OWASP, http://www.owasp.org/index.php/Data_Validation