Web Application Worms: Myth or Reality?

Automated, self-propagating attacks on vulnerabilities in custom Web application code

Written by
Amichai Shulman
Chief Technology Officer

IMPERVA™
APPLICATION DEFENSE CENTER

Table of Contents

Abstract

This paper discusses the possibility of automated, self-propagating attacks on custom Web application code. It will show that such attacks are not only feasible but that their theoretical success rate is far greater than worms targeting commercial infrastructure (e.g., Slammer, Code Red, Blaster, Nachi, etc.).

It is the intent of this paper to raise awareness of the threat posed by automated attacks on vulnerabilities that exist in every organization's Web infrastructure. Threat's of this type that cannot be avoided by counting on current IPS technologies and the law of large numbers.

Introduction

Application level attacks are attacks on the top layer of the OSI model – the application layer. These attacks may target either generic application infrastructure solutions, (e.g. IIS or Apache) - or they can attack the custom code and business logic which is unique to each Web application. Traditional automated worms are commonly designed to exploit known vulnerabilities of generic infrastructure solutions. A worm enables a single hacker to simultaneously attack a multitude of Web sites.

Attacks on custom code normally use manual techniques such as SQL Injection, parameter tampering, forceful browsing, etc to exploit vulnerabilities that are unique to each web application. For example, the SQL injection may be used to manually exploit a vulnerability that exposes credit card numbers in Acme Corp.'s ecommerce application. SQL injection may also be used to manually exploit a completely different vulnerability that exposes account balances in BankX's online banking application. Although general SQL Injection process is the same in each case, the specific vulnerability and the steps required to build the exploit are completely unique. Hence the following convictions are common among IT decision makers.

- Identifying custom application vulnerabilities within a site requires "personal attention".
- Web application vulnerabilities from one custom web application cannot be reproduced to another
- Finding vulnerabilities in custom Web application code and writing successful exploits is difficult and requires advanced hacking skills. Hence the number of actual hackers that can practice application hacking is small.

As a consequence, most organizations conclude that unless they own a very high profile application with very high potential profit for

hackers, the chances to be hit by custom code attacks are very slim. Since those organizations face a daily battle with infrastructure worms such as Slammer, Blaster and others, they tend to focus their attention and resources on security solutions that can stop this type of mass attack.

The next section discusses automated, self-propagating attacks on custom Web application code. It shows that the technology required for creating such an attack is highly accessible, that the skills required are common, and that the potential proliferation rate among valuable targets is higher then that of commercial infrastructure worms. It also shows that is does not matter whether you are a prominent American bank or a small e-vendor in Poland, you have the same chance of being hit by such an attack. A feasibility study conducted by the Imperva ADC demonstrates the validity of these facts.

Anatomy of an Automated Application Worm

For the purpose of our discussion we will describe an imaginary application level worm called Niddhog[1]. In order to be a healthy and prosperous worm Niddhog must have the following capabilities.

1. An efficient method of finding its prey. Since Niddhog is an Web application worm the prey must be an identified web site rather than an IP address.

2. A method for identifying vulnerabilities in potential prey. These must be specific URLs or even specific parameters within URLs.

3. A method for exploiting such vulnerabilities in a way that allows Niddhog to deploy its Trojan horse (copying the payload and Niddhog code to the victim site).

4. A method for activating the attack and the new copy of the code. Since this is an application level worm we would expect the Niddhog to activate the new copy by simply making an HTTP request to a URL on the exploited server.

Conventional worms use random address generation for the first task, the law of large numbers regarding a specific known vulnerability for the second, and hard-coded exploit code for the third. Niddhog will instead use a special technique that efficiently finds vulnerable sites and specific vulnerabilities in a single step. This technique, that we call "**War Searching**", will be explained in the next section. Using this technique Niddhog is assured of having far less failed attack attempts than conventional worms, leaving a much

[1] In northern mythology a worm that gnaws the roots of the world-tree.

smaller footprint in network traffic, and taking much less time to achieve massive proliferation. This, in turn, assures that it can achieve more damage prior to any mass protection schemes are deployed to stop it.

Creating an exploit is usually a straight forward task once vulnerability has been discovered. However, some special tricks may be required for application level attacks. These tricks are explained in the **Advanced War Searching** section of this paper. Finally launching the exploit code or activating it can be achieved through a technique we call “Search of Death” in the final section of this paper. Using this technique the attacker can almost perfectly cover his or her tracks.

War Searching

Search Engines

The ideal place for an application level worm to look for potential victims and their vulnerabilities would of course be a directory that lists all applications (by name) and their vulnerabilities (preferably by type). It turns out that such a directory exists. Those are the search engines and Web directories such as Google, Yahoo, Altavista, MetaCrawler, etc.

Search engines use a network of computers to continuously map the contents of Internet Web servers using a species of software program that has earned the nickname WebBots. WebBots repeatedly and methodically crawl the Internet link, to link, to link. Once a link is followed the contents of the reply are indexed by the search engine and the robot follows any links found within that reply.

A search engine may start mapping a given Web site either per an explicit request (See http://www.google.com/addurl.html for an example of such a request) or by following links to the site from other sites traversed by the WebBot. However once a WebBot starts mapping a site, it cannot be stopped. GoogleBot (Google’s WebBot) has become so thorough that it can even trace links created through JavaScript code in HTML forms. As a consequence, once a site has been “discovered” by a search engine, the WebBot operating on behalf of that search engine will discover all publicly available URLs at the site including URLs that were not intentionally exposed to the public. After the WebBot has indexed a site, the indexed URLs are publicly available and can be retrieved using keywords from both URL and content of the reply.

Reconnaissance

Enter the “War Searching”, which we define as an automated compilation of vulnerable sites with specific vulnerabilities by using an Internet search engine. The basic idea supporting this method to exploit an individual site has been discussed in various public forums since 1991. However, it has not yet been noted that these same ideas can be used as the basis of an automated attack against numerous sites. Let’s examine some simple examples of War Searching and the results they yield.

A common example among early War Searchers is the search for the term “Select a database to view” which currently yields approximately 600 results. Some of them link to pages where a list of FileMaker WebCompanion files can be found and accessed directly. Adding the term “FileMaker” to the search narrows result to 450 entries of which the vast majority are actual links to sites that expose FileMaker companion files (the others discuss this possibility). Such files may contain sensitive raw data that is unintentionally exposed to direct access. Another example the search for the terms “index of /etc” and passwd. This yields 270 results - most of which are links to pages that link directly to an unprotected password file.

So the idea is to come up with the right combination of terms that would yield the appropriate results regarding some vulnerability. Our , Niddhog worm, for example, might search for the term +“index of” +service.pwd yielding a result set of approximately 650 links for FrontPage extension password files (an ancient vulnerability). One may argue that this is a small amount of servers not worthy of a mighty worm like Niddhog, but we must remember that almost every result yields an vulnerable site. Traditional application infrastructure worms, on the other hand take the approach of generating addresses at random, validating their existence, and then hoping that they host a server of the right type with the right vulnerability. To match Niddhog’s results of 650 vulnerable sites, this traditional worm would require an average of 6.5 million failed attempts to find the first vulnerable site. A more prominent vulnerability that is found in 100,000 Internet sites (an incredibly large number) would still take approximately 15,000 failed attempts by a traditional infrastructure worm to find the first vulnerable site.

Since search engines return the result set as an HTML document, it is easy to write code that would extract the vulnerable URLs from the reply. In fact, Google now exposes a Web Service interface that makes such a task even easier. Hence the attacker has a simple piece of code that uses HTTP requests to retrieve vulnerable URLs from a search engine. By making small changes to the code, Niddhog can use a different search engine or look for a different vulnerability. It is important to note that this discovery stage does not require any direct interaction with the target site. Hence, no protection mechanism at the site could set off an alarm.

Attack

The next step for Niddhog would be to traverse the list of potential targets and extract the actual password file from each of them. It's follows that since the WebBot can access the URLs in the result set (or else they would not be indexed) so can Niddhog. No protection mechanism stands in the way of such access. After retrieving the password files, Niddhog uses freely available password cracking software (e.g. John the Ripper, L0phtCrack, etc.) to extract a list of clear text passwords. Numerous studies have shown that the success rates of these tools are extremely high requiring a relatively short amount of time.

To complete the attack Niddhog would create a FrontPage http request to each of the sites in the results set, using the appropriate credentials and uploading the Niddhog code to the site. A simple ASP page is uploaded that, (when accessed) invokes Niddhog on the remote computer. A more sophisticated attack would make the ASP page a bit more stealth (e.g. return HTTP code 404 unless a password is given) and the Niddhog code configurable in a way that each exploited server searches only a predefined portion of the result set. It should be noted that the HTTP requests used by Niddhog to actually attack the site are well-formed, legitimate requests and that the URLs used by Niddhog are legitimate URLs (or else they would not be indexed). Hence, no existing Intrusion Prevention System (IPS) or firewall would be aware of the attack.

Advanced War Searching

The previous section showed a very simple example of War Searching. Our research also demonstrated the viability of application level worms that have a potentially high yield but require more advanced War Searching techniques

Narrowing the Search Space

Assume that an attacker wants to exploit a vulnerability in a content delivery package (lets call it BeContent for this discussion) that relates to exposing a password file called "password.txt". The location of such a file would depend on the tree structure of each site deploying the content delivery package. Searching for "password.txt" would yield an immense result set, most of which is irrelevant to the exploit. However, the attacker knows that a site deploying the BeContent package will expose URLs of the form /<somepath>/BCCDeliver/<somefile>. They would then create a worm to search for the term BCCDeliver and then search again using the "site:" option to obtain the location of password.txt file for each site obtained from the first search.

Finding SQL Injection Vulnerabilities in Custom Code

Assume that the attacker would like to create a worm that exploits SQL injection vulnerabilities in custom code to inflict denial-of-service on a site. Creating such an attack would require identifying specific points in the business logic of that Web application that are susceptible to SQL injection attacks.

The attacker can use a search engine to search for the term [Microsoft][ODBC SQL Server Driver][SQL Server] which yields approximately 130,000 results, a large portion of which are pages that suffered a database access error when visited by the WebBot. The attacker can make the search request focus further on each site by repeating the same query but with the “site:” option and collecting all potential vulnerabilities within a site.

Discovering the vulnerable URLs is not enough for successful SQL injection. The attacker must further find the parameters that are susceptible to SQL injection. This is achieved by again using the “site:” option, this time using the vulnerable URL as the search term. The result set includes references to the vulnerable URL from which we can extract the parameters. No special hacking or programming skills are required.

To complete the attack, the hacker must invoke one of many tools that can be obtained from the Internet that is capable of efficiently achieving actual SQL injection given a URL and its parameters. It should be noticed that this last step may leave an apparent footprint in the attacked server logs, since a substantial number of attempts is usually required by the automated tools to achieve SQL injection. The attack code can cover its track and evade being traced back to its source by using the “Search of Death” technique described in the final section of this paper.

Using Internal Site Search Capabilities

Some resources may not be available for external indexing either because the webmaster bothered to create a robots.txt file or because they were not properly linked to the rest of the site. In some sites, such resources, as well as other internal resource may have been indexed by an internal indexing tool such as MS Indexing Service. Such services index the site by traversing the directory tree through the operating system to yield interesting results. An attacker can use a search engine to find the internal search form of sites (e.g. by looking for the term search.asp) and recognizing the parameters it requires (see above). They can then use this internal search engine much the same way as they could use a public search engine.

The Search of Death

Most attackers with malicious intent prefer to remain anonymous. The author of Niddhog would be no exception. One obvious way to launch Niddhog anonymously would be to manipulate the WebBot into attacking vulnerable sites.

At some point the attack must directly access the victim's site. This could be the final stage of the attack in which the Niddhog code is uploaded to the attacked server and activated, or it could be an earlier stage in which SQL injection strings are being constructed. In order to avoid detection and tracing of the attack source, an attacker would usually break into an intermediary computer and launch the attack from it. However, Web application attacks may take advantage of another option that does not involve the hacking skills required for breaking into an intermediary server. If the attack can be reduced into a single URL, (and most application attacks can be), then the attacker can use a technique that we call the Search of Death (SoD).

The SoD is defined as the use of a search engine as a proxy of the real hacker to launch an application attack against a chosen target. Implementing SoD is straightforward. The hackers creates their own anonymous Web site, using one of many available free hosting services. They then submit the site to a search engine for indexing. When they observe that the search engine paid them a visit (e.g. by inserting rare terms within the content of the site and searching for them in the search engine) they create a new page which contains the attack URLs. The next time the WebBot pays a visit it will follow the links in the new page and index the results. Following the links in the malicious page means that the WebBot will launch the attack URL against the target site. If this URL is the final attack then the hacker's job is done. If they are using the attack URL for further information gathering, then they need to search using the "site:" option to read the reply to their attack URLs. If an attacked site detects the malicious request, all tracks lead back to the WebBot.

Conclusion

Application level vulnerabilities in commercial infrastructure software and custom Web application code are common in today's Internet infrastructure. Using the War Searching techniques defined herein, an attacker can efficiently identify a multitude of such vulnerabilities

in a way that guarantees a very high attack success rate. This success rate is 4 to 7 orders of magnitude larger than that of application infrastructure worms guessing their way across the Web. Moreover, throughout most of the discovery stage, the victims are completely unaware of the attacker's activity.

Using War Searching to identify vulnerable sites guarantees that even if the attacker seeks to exploit a long ago published vulnerability (such as our legendary Niddhog worm) for which countermeasures have long been available, a multitude of servers remain vulnerable.

Once the attack is launched, its proliferation rate is guaranteed to be very high. In fact, by the time anybody identifies such an attack, most vulnerable servers connected to the Internet would have already been compromised regardless of how well their traditional protection devices are configured.

Finally, we have shown that the initial introduction of the worm to the Web as well as further attacks by contaminated victims can be achieved through the search engine's WebBot. Using this technique makes the source of such a worm virtually untraceable.

We therefore conclude that the accepted notion that application vulnerabilities in custom code are a tertiary risk due to the allegedly manual nature of application attacks on custom code is faulty.

Bibliography

- "Watching the Watchers: Hacking with Google", 2002
  http://johnny.ihackstuff.com

- “Google not ‘hackers’ best friend”, James Middleton, VNUnet.com, 2001
  http://www.vnunet.com/News/1127162

- Google: Net Hacker Tool du Jour, Christopher Null, wired.com, 2003
  http://www.wired.com/news/infostructure/0,1377,57897,00.html

- “Watch out, Google”, Sebastian Wolfgarten, 2003,
  www.wolfgarten.com/downloads/Watch_out_google.pdf

- Google: A dream come true, Comsec, 2003,
  www.governmentsecurity.org/comsec/googletut1.txt

Imperva, Inc.
12 Hachilazon st. Ramat-Gan **52522**, Israel
Tel: **+972 3 6120133** | Fax: **+972 3 7511133**
U.S. Toll Free: **1-866-592-1289**

1065 East Hillsdale Blvd., Suite 109
Foster City, CA 94404, USA
Tel: (650) 345-9000 | Fax: (650) 345-9004

info@imperva.com | www.imperva.com