WHITEHAT™ SECURITY

AUTOMATED SCANNING VS. THE OWASP TOP TEN

JANUARY 2007

Jeremiah Grossman

Founder and CTO, WhiteHat Security

A WHITEHAT SECURITY WHITE PAPER

The OWASP Top Ten[1] is a list of the most critical web application security flaws – a list also often used as a minimum standard for web application vulnerability assessment (VA) and compliance. There is an ongoing industry dialog about the possibility of identifying the OWASP Top Ten in a purely automated fashion (scanning). People frequently ask what can and can't be found using either white box or black box scanners. This is important because a single missed vulnerability, or more accurately exploited vulnerability, can cause an organization significant financial harm. Proper expectations must be set when it comes to the various vulnerability assessment solutions.

For our part, WhiteHat Security is in the website security business and provides a vulnerability management service. Our Sentinel Service incorporates expert analysis and scanning technology. Using a black box process, we assess hundreds of websites a month, more than anyone in the industry. What we've come to understand is a significant portion of vulnerabilities are virtually impossible[2] for scanners to find. By the same token even the most seasoned web security experts cannot find many issues in a reliable and consistent manner. To achieve full vulnerability coverage we must rely on a combination of both methods.

We'd like to share some of our experiences that lead to this conclusion. Using situations we've seen in the real world, and the OWASP Top Ten as a baseline, we'll demonstrate why scanning technology alone cannot find the OWASP Top Ten. To begin we'll focus on a single feature of a fictitious Web Bank responsible for funds transfers from one account to another account. Here is the full URL:

http://server/transfer.cgi?
from_acct=1235813&to_acct=31415&amount=1000.00&session=1001

The "from_acct" is the current user's account number. "to_acct" is where the money should be sent. "amount" is obviously the transfer amount, and the "session" is the authenticated session ID after having properly logged-in. This is a fairly typical and straightforward business process.

A1 Unvalidated Input

Scanners must hazard a guess about what "transfer.cgi" does. Otherwise it would be impossible to determine what it should NOT do.
Humans can easily figure this out, but scanners aren't that intelligent. There is no knowledge of context. For the sake of discussion, let's say a scanner has the ability, because there's a dollar figure present and "transfer" keyword in the URL might help it decide that this feature moves money. Realistically, these parameter names could be anything and often far more cryptic. To attempt a classic funds transfer attack, let's change the above URL substituting the "1000.00" amount to "-1000.00."

---

[1] OWASP Top Ten
http://www.owasp.org/index.php/OWASP_Top_Ten_Project

[2] undecidable problem
http://www.nist.gov/dads/HTML/undecidableProblem.html

Negative Amount Example:

http://server/transfer.cgi?
from_acct=1235813&to_acct=31415&amount=-1000.00&session=1001

By transferring a negative amount, the web application would potentially deduct money from the target account instead of adding to it! The challenge for a scanner is being able to decide whether or not the attack succeeded. How can it tell?

If the fraudulent transfer succeeded the website might respond with, "Success, would you like to make another transaction?," "Transfer will take place by 9 am tomorrow," "Request received, thank you," or any number of possible affirmations. If the attack failed, "Transfer failed", "Error: Transfer amount must be a positive number", or "Bank robbery detected, men with guns have been dispatched to your location!" Every custom Web Bank application will likely respond in a different manner. That's the problem! Pre-programming in all the possible keyword phrases or behavioral aspects is simply unfeasible and for all mathematical provability, impossible. However, human gray matter can make this determination.

Let's put this into perspective: A scanner might be able to figure out what the transfer.cgi feature is or does. It may be capable of generating a properly formatted fraudulent transfer in order to test for a potential security issue. And, it is potentially smart enough to discern if the attack worked or didn't through a keyword/behavioral match. Sometimes this process works, more often it doesn't. Now, take into account the number of features on a website, and that there are over a hundred million websites[3] out there in the world. You then start to get an idea of the complexity of the problem. The point is if your website MUST be secure this is a lot of "might", "maybes", and "potentially's" to depend on.

A2 Broken Access Control

Broken access control is a common flaw in web applications that allows attackers to execute functionality without proper authorization. For example, the Web Bank may allow a funds transfer from someone else's account. To do this test, we'll swap the account numbers in the "from_acct" and "to_acct". If the web application forgets to check if we're authorized make such a request, we'll be in business.

Original:
http://server/transfer.cgi?
from_acct=1235813&to_acct=31415&amount=1000.00&session=1001

Unauthorized Account Example:
http://server/transfer.cgi?
from_acct=31415&to_acct=1235813&amount=1000.00&session=1001

[3] Netcraft November 2006 Web Server Survey
http://news.netcraft.com/archives/2006/11/01/november_2006_web_server_survey.html

This is a simple example of an unauthorized funds transfer attack, because the web application lacks a simple business logic check. From a vulnerability scanner perspective, this is a nearly identical problem to "Input Validation". How does a vulnerability scanner know what this feature does? How does it know to swap these numbers or what that might achieve? Or, what does the attack look like when it worked? For a human sitting in front of a web browser it's a piece of cake. For a scanner it's exponentially harder. The bottom line is sometimes scanners uncover similar issues; we're just never exactly sure when the scanner's results are accurate or not.

A3 Broken Authentication and Session Management

Websites require session IDs to uniquely track users as they traverse the site. Without session IDs a user would have to enter their username and password with every link click. Session IDs may be stored in cookies, post data, or - in our example a URL parameter named "session." To prevent against Session Hijacking, an attacker falsifying another user's session ID, the identifier must be unguessable. Session IDs should be sufficiently random, long, and assigned after a user has properly logged-in. Poorly coded web applications often use predictable incrementing numbers. Successive log-ins will generate session IDs of 1001, 1002, 1003, and so on.

To perform a Session Hijacking, an attacker would properly log-in to the website, manually decrement this number in the URL, and then check the results. If the results seem promising, an attacker may spend several hours or days on this process, depending on their patience and strength of the system.

Original:
http://server/transfer.cgi?
from_acct=1235813&to_acct=31415&amount=1000.00&session=1001

Session Hijack Example:
http://server/transfer.cgi?
from_acct=1235813&to_acct=31415&amount=1000.00&session=1000

For vulnerability scanners, this is a multi-faceted problem. A scanner must first learn how to login, which often requires human intervention. Second, it must locate the session ID in order to start analyzing its relative predictability. (Scanners are great at this number crunching phase.) Lastly, a scanner must decide if changing session IDs results in landing inside another user's account.

This is another one of those "did it work?" questions from the last two examples. What is the web page supposed to look like when logged-in as one user versus another? They could be nearly identical, vastly different, or somewhere in between. This gray area is where false-positives and false-negatives live. Current state-of-the-art scanning technology can reduce the time it takes to complete this part of the vulnerability assessment process, but we're far from a system that can detect Session Hijacking issues on even some of the websites some of the time.

A4 Cross Site Scripting

Cross-Site Scripting (XSS) has swiftly become information security's most common vulnerability. Every day we learn more about this once underestimated vulnerability. The good news is that vulnerability scanners are very adept at finding this issue and happen to be one of our more accurate checks. The bad news is that we're constantly finding new places where XSS can show up and new ways to get it there that scanners have a very hard time detecting.

When data is echoed into HTML space without any filtering, that's a no-brainer for XSS detection. What we're seeing more often is good filtering, but the user-supplied input is dropping inside JavaScript space. This means an attacker wouldn't necessarily need to filter < and > to execute JavaScript. Often it's enough to have quotes, semi colons, and parentheses. So with XSS detection, now not only do we need to know what data is making it through the filter, but also in what context it's landing on the following page. For scanners, the process isn't perfect, but pretty good.

A5 Buffer Overflow and A6 Injection Flaws

For more than a decade, buffer overflow exploits and other types of injection flaws have been used in countless attacks, viruses, worms, and other forms of malware. A buffer overflow exploit works by writing more data into a memory buffer than it's allocated to hold, with the intent of altering the flow of the application. Normally, these issues are found through source code analysis or by reverse engineering binaries. This is why buffer overflows usually are discovered within compiled commercial and open-source software, rather than in custom web application code[4].

In custom web applications, penetration-testers often have zero access to source code, application binaries, or memory dumps. Testing is blind. A typical test of a custom web application will input several thousand characters into a URL query string parameter and see how the web server reacts. Let's try loading a thousand A's into the session parameter value.

Buffer Overflow Example:
http://server/transfer.cgi?
from_acct=1235813&to_acct=31415&amount=1000.00&session=AAA ... 1000

If the web server responds with a 500 response code (Internal Server Error) this may be an indication that an application crashed due to a buffer overflow, but that is extremely unlikely. More likely a device in the connection chain, including web application firewalls, HTTP proxies, load balancers, web servers, application servers, or others, caught the abnormal request and returned a 500 error message. This is why blind buffer overflow tests are prone to heavy false-positive rates. Anything could have caused a 500 error, but only rarely is it a web application buffer overflow. As before, sometimes a vulnerability scanner can spot this vulnerability, sometimes not.

---

[4] Myth-Busting Buffer Overflows
http://www.whitehatsec.com/home/resources/articles/assets/mythbusting_buffer_overflow.pdf

The other challenge is popular web application programming languages such as Java, Perl, PHP, Ruby and others are immune to buffer overflows in the first place. Be skeptical of any report that says a buffer overflow exists in an application written in one of these languages.

A7 Improper Error Handling

ODBC error messages, Java stack traces, and template errors are all issues we've become accustomed to seeing in vulnerability assessments and everyday web surfing. For the most part, it's easy to pre-program scanners with the most common error message patterns that reveal a little too much information. Other error messages we encounter are subtler.

For instance, an attacker could purposely use incorrect username and password login attempts to figure out which half was wrong. This is an effective technique for breaking into accounts or discovering which users are registered. In a similar way, password recovery features can be abused when they ask for user email addresses. By guessing an email address the system may disclose if it exists or not on the system, which spammers enjoy. There are endless examples, but let's attempt one from our Web Bank transfer feature.

By rotating, brute forcing, or guessing our way through "to_account" numbers we might be able to get the system to tell us if those accounts exist on the system. Careful, we don't want to send any money during these transactions. We need to change the "amount" to "0.00". If we'd forgotten, we would have run out of money before testing was complete!

Original:
http://server/transfer.cgi?
from_acct=1235813&to_acct=31415&amount=1000.00&session=1001

Account Rotation Examples:
http://server/transfer.cgi?from_acct=1235813&to_acct=31416&amount=0.00&session=1001

http://server/transfer.cgi?from_acct=1235813&to_acct=31417&amount=0.00&session=1001

http://server/transfer.cgi?from_acct=1235813&to_acct=31418&amount=0.00&session=1001

In response to these requests, the website could react in any number of ways. The website could respond with an error message, "account number X does not exist, please check the number and try again". That's revealing. Or, what if it said, "Transfer waiting to be processed"; meaning you have no idea what has taken place on the backend. Or, maybe the website logs you off entirely, alerting the system administrators, because you're trying to scam the system. These vague possibilities are extremely hard for any scanner to understand in a generic fashion. Sometimes it will know, but most of the time it won't. It depends on the website and its developers.

A8 Insecure Storage

Sensitive information flows in and out of websites in a variety of ways and locations. Sometimes it's done securely using trusted crypto algorithms. Other times sensitive data is

just encoded, or worse, passed in plain text. Or, sensitive data is stored in a backend database only indirectly interacted with, but from the outside we have no idea how it's stored. Once something like a credit card number is handed to a website, outsiders have no idea if it's stored in an encrypted form or not. And, right there, from a scanner perspective, how would it ever know! In fact, a human would never know either. This is a huge lack of visibility from an outside viewpoint.

A9 Application Denial of Service

Websites are complex creations with many layers of software and services that depend on one another. Astonishingly, even the most popular websites can be incredibly fragile and susceptible to several forms of application Denial of Service (DoS) attacks. Sometimes breathing on them wrong causes a website to fall over. One easy DoS attack is against search fields. Typing in a dozen or so terms separated by "ANDs" simultaneously can be very taxing on a CPU. (example: "searching AND for AND this AND term AND that AND term AND requires AND a AND lot AND of AND cpu AND time".) Or, if a login system has anti-brute force features, an attacker could purposely guess usernames to lock out legit users system-wide. Customer service really dislikes it when this happens.

A penetration tester might inadvertently cause something similar to occur in the Account Rotation example from A7 Improper Error Handling. The tests could have generated thousands of ghost transactions in the system, each taking a little while to complete. Too many transaction requests and the system may leave legitimate transactions waiting in a huge queue. This brings us to one of the biggest problems with testing for DoS attacks. The last thing you want to do is impact business continuity in either staging or production systems. A good penetration-tester will document a DoS as theoretically possible and ask a developer to double check the code. The point is scanners don't really know if they're causing harm, experts need to be watching over the process to keep things safe.

A10 Insecure Configuration Management

All major web server and application server platforms from Apache and Microsoft IIS to JBoss and IBM WebSphere have configuration files. Many of these files will impact website security and should not be disclosed publicly. Normally, these files are located in inaccessible areas because they might contain sensitive information. Occasionally a Java web.config or Apache's httpd.conf can be found hanging out on a public web server– not too often, but it happens. In most cases, to determine if a website is properly configured for security, you need to send a series of requests and then analyze the way certain web servers react. You might be able to tell what software version the web server is running or what features are enabled. Other aspects like process or directory permissions are not so easy to detect from a remote location. Again, like the other issues in the OWASP Top 10, we can scan for some, others we can't.

Conclusion

Websites and web applications come in all different shapes, sizes, and complexities. And, just like any piece of software they come with their own unique set of bugs. Software has bugs, and hence will have vulnerabilities, we all know and accept that.

The key to avoiding being exploited is finding and fixing all the vulnerabilities on your corporate websites. Automated scanning technology can help in the process, but as we've seen in the examples above, scanners alone cannot identify the entire OWASP Top Ten.

To be effective, the goal of a website security program should be to find everything. It only takes one flaw to allow an attacker to victimize your company or your customers. Human evaluation is an integral and necessary part of the vulnerability assessment process. WhiteHat Sentinel combines the best scanning technology and expert analysis to provide complete website vulnerability management.

WhiteHat Sentinel reduces the burden of securing websites with an ongoing service that provides up-to-date and comprehensive identification of the vulnerabilities that are putting online customer and corporate data at risk. It is the only solution that can assess for all 24 classes of vulnerabilities identified by the Web Application Security Consortium's (WASC) threat classification.

At WhiteHat, we're constantly developing new technology to ensure the integration of tools and people, improve efficiency, consistency, and repeatability of the assessment process to keep pace with the latest attack vectors.

About WhiteHat Security, Inc.

Headquartered in Santa Clara, California, WhiteHat Security is a leading provider of website security services. WhiteHat delivers turnkey solutions that enable companies to secure valuable customer data, comply with industry standards and maintain brand integrity. WhiteHat Sentinel, the company's flagship service, is the only website vulnerability management service that incorporates expert analysis and industry-leading technology to provide unparalleled coverage to protect critical data from attacks. For more information about WhiteHat Security, please visit our website, www.whitehatsec.com.

FOR MORE INFORMATION ABOUT WHITEHAT SECURITY, PLEASE CALL 408.492.1817 OR VISIT OUR WEBSITE, WWW.WHITEHATSEC.COM