4th USENIX Workshop on Offensive Technologies (WOOT '10)

August 9, 2010
Washington, DC

VULNERABILITY ANALYSIS

No reports are available for this session, which included the following paper and invited talks:

- ***All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)*** (Invited Talk)
  Edward J. Schwartz, Thanassis Avgerinos, and David Brumley, Carnegie Mellon University
- ***Zero-sized Heap Allocations Vulnerability Analysis***
  Julien Vanegue, Microsoft Security Engineering Center
- ***Beyond Heuristics: Learning to Classify Vulnerabilities and Predict Exploits*** (Invited Talk)
  Mehran Bozorgi, Lawrence K. Saul, Stefan Savage, and Geoffrey M. Voelker, University of California, San Diego

CRYPTOGRAPHY, ETC.

Summarized by Adam J. Aviv (aviv@cis.upenn.edu)

- ***Recovering Windows Secrets and EFS Certificates Offline***
  Elie Burzstein, Stanford University; Jean Michel Picod, EADS

Elie Burzstein discussed the Windows Data Protection API (DPAPI), a "black box" for encrypting and decrypting data that is used in many different parts of the Windows operating system, including the Encrypted File Systems (EFS), as well as a variety of other programs (Skype, Explorer, WiFi, etc.). Burzstein provided key insights into mounting the Windows EFS on Linux. This work also shows how one may perform a key escrow attack on the DPAPI to achieve this goal.

In the first part of the presentation, Burzstein introduced the DPAPI in great detail. Moving quickly, the talk covered the ins and outs of key management and the various structures that store and implement encryption. Some of the more important points are that the keys used by the DPAPI are seeded with a hash of the user's password, and keys renew every three months (and when the password changes).

The current keys are stored in the %APPDATA%, a Windows protected file, inaccessible by outside applications or operating systems. Still, Windows must know which block is encrypted with which key, and to do that, timestamps are used. This is where things get interesting. Burzstein et al. noticed that the timestamps can be altered to prevent key renewal. However, there is still the pesky password-changing problem, but the master key describes the current password hash. A password chain is used, so by using the current password it can decrypt all previous encrypted blobs.

Burzstein could not demo his tool because he was presenting from a Mac. He did outline some goals: to make this work on Windows 7 and to look at retrieving the password from non-volatile memory. "Questions?" he asked in conclusion. "The best part of the talk is I never get questions."

- ***Crawling BitTorrent DHTs for Fun and Profit***
  Scott Wolchok and J. Alex Halderman, University of Michigan

In a fascinating presentation, Scott Wolchok discussed crawling the BitTorrent DHT (for fun and profit). This work is closely related to Wolchok's widely publicized work on Un-Vanish (NDSS '10) where he crawled the Vuze distributed hash table (DHT) to defeat Vanish (Sec '09). In this work, Wolchok concerned himself with the primary purpose of the Vuze DHT: to catalog BitTorrent (BT) meta-information, and what can be done with information collected from crawling this data.

Wolchok began his talk by noting that torrent-tracking Web sites are under legal attack because of their centralized nature. As a result, distributed and decentralized tracking services are quickly becoming the norm. Such services make use of DHTs, and the ability to crawl the DHT to collect torrent information could be seen as a defense against legal attack. Although a torrent site may be taken down, a single, overnight crawl of the DHT provides enough information to rebuild the BT site. Conversely, the same crawl also reveals a large amount of information about the users who download torrents, which may be used to file lawsuits—fun and profit.

Perhaps the most interesting part of the talk was when Wolchok presented the results of his crawls with respect to the variety of different torrents seen: "Everything in the top seven infringes copyright. It isn't used to download Linux ISOs." The top 1,000 torrents were also "not obviously" copyright neutral, and most torrents tended to be fairly recent TV and movies. Wolchok bemoaned that he had to stop the crawl prior to the *Lost* finale (having only done a single crawl, resulting in an estimated 20% coverage). The most recent *Lost* torrent was one of those popular torrents and its activity seemed to spike on Friday night and Saturday morning. He offered one explanation: "Pirates have jobs too," and probably don't get around to downloading the show until the end of the work week.

- ***Practical Padding Oracle Attacks***
  Juliano Rizzo, Netifera; Thai Duong, VNSECURITY

Juliano Rizzo demonstrated how he and his co-author, Thai Duong, performed online attacks by altering the CBC padding in captured blocks of ciphertext (first presented by Vaudenay at Eurocrypt 2002). The key observation of Rizzo and Duong is recognizing that *padding oracles* are everywhere on the Internet, which allows an attacker to crack encrypted cookies, CAPTCHAs, and much other encrypted content. By slightly altering the padding bits of the encrypted blocks sent to the oracle, a response of either "Invalid" or "Valid" is enough to decrypt one byte of the ciphertext.

Repeating the process, the message is incrementally decrypted, back to front.

The most exciting part of the presentation was when Rizzo played some videos of his padding oracle in action. In the first demo, an encrypted cookie from a JavaServer Faces client was incrementally decrypted. In the second demo, Rizzo showed how this attack can be used to break a CAPTCHA. The text of the CAPTCHA entry is encrypted with the CAPTCHA image, and the padding oracle is the CAPTCHA server. Slight alterations of the padding region and a useful error message from the server ("PADDING ERROR") are more than sufficient to decrypt the CAPTCHA text. Again, the video demo presented was very cool, and slowly but surely the text of the CAPTCHA was revealed within the tool's display region. (The video included a text flash "10 minutes later," which got chuckles from the audience.)

THE WEB AND SMARTPHONES

Summarized by Scott Wolchok (swolchok@umich.edu)

- ***Busting Frame Busting: A Study of Clickjacking Vulnerabilities on Popular Sites*** (Invited Talk)
  Gustav Rydstedt, Elie Bursztein, and Dan Boneh, Stanford University; Collin Jackson, Carnegie Mellon University

Collin Jackson opened by explaining that frame busting refers to JavaScript code that Web sites use to prevent themselves from being framed, and that Web sites typically don't do frame busting very well. He explained that frame busting code typically consists of two parts: a conditional statement intended to detect framing, and a counteraction intended to remove the framing or disable the page. Frame busting is intended to defend against several attacks, including clickjacking, where attackers overlay a benign-looking page intended to trick a user into performing some action on a victim page (e.g., deleting a Twitter account), and attacks on per-site images attempting to authenticate a site to the user as a phishing defense, which can be defeated by framing the victim site's login page to display the image.

Jackson then presented the results of the authors' survey of frame busting code on the top 500 Web sites according to Alexa. They found that frame busting was very common on the top 10 Web sites (60%), but not so common on the top 100 (37%) and top 500 Web sites (14%). They also observed that frame busting code was very diverse, with at least 10 different conditional statements and even more different counteractions, and Jackson claimed that every site in the top 500 had broken frame busting code. He elaborated on problems with specific sites, most of which revolved around attempts to allow framing from certain referrers.

Next, Jackson covered a variety of other attacks on frame busting code. Location clobbering attacks browser bugs that allow a framing site to mask top.location to prevent the framed side from detecting framing. The attacker can also "ask nicely" in JavaScript to get the user to cancel the framed page's redirection counter-action and can cancel the navigation programmatically by overloading the browser with 204 No Content responses. Several browsers allow disabling JavaScript in an iframe, and reflected XSS filters in IE8 and Chrome can be abused to remove frame busting scripts as well.

Jackson closed by discussing mitigations to the frame busting problems he presented. A pair of HTTP headers, X-Frame-Options and Content Security Policy, allow sites to control framing at the HTTP level, although they have tradeoffs in terms of complexity and flexibility. The authors put forward a new JavaScript frame busting defense that "fails safe" by rendering the document invisible if it is unable to frame bust.

Rik Farrow asked where the defense code could be found, and Jackson replied that it is located at http://seclab.stanford.edu/websec/framebusting/. Someone asked Jackson to clarify whether JavaScript had to be mandated to protect against frame busting attacks. Jackson stated that this is somewhat true and very controversial, as many people think that JavaScript is evil and a security problem. Jackson said that he is skeptical of solutions that try to lock down Web features, although he recognizes that some people may want to. He stated that the X-Frame-Options header works with JavaScript disabled. Someone else asked how to protect a user like his mother, who would click everywhere on the invisible document generated by the authors' frame busting code in an attempt to fix the "problem." Jackson responded that the code sets the display:none CSS property on the body element, which prevents click events.

- ***Smudge Attacks on Smartphone Touch Screens***
  Adam J. Aviv, Katherine Gibson, Evan Mossop, Matt Blaze, and Jonathan M. Smith, University of Pennsylvania

Adam Aviv opened by summarizing the authors' work: "I took a lot of pictures of smartphones with smudges on them." He presented several examples of forensic information leakage, including taking a rubbing from a pad of notepaper, wear patterns and residual heat on keypads, and residual fingerprints on a touchscreen. He explained that the authors' work focused on smudges left on Android phone touchscreens after performing the password wipe sequence to unlock the phone. In the wipe sequence, the phone shows a grid of nine points, and the user must trace a line through several of them. Points can neither be skipped nor reused. Aviv observed that the pattern space is fairly small; it consists of 389,112 patterns, and a similar PIN entry space (4–9 digits, used once) contains over 1 million passwords.

Next, Aviv explained the experiments that the authors performed. They considered one particular swipe pattern touching all nine dots and selected to provide several different directions. The swiped phones were photographed while varying the lens angle and the vertical angle to the phone. The photographs were classified on a scale from 0 to 4,

where 4 was fully observable and 1–3 were partially observable with the loss of one or more directions.

The first experiment dealt with determining the angles and lightings that provided for ideal collection of smudges and used four smudge configurations: an HTC G1 with "normal" touches, "light" touches, touches with facial contact, and a Nexus One with "normal" touches. After Aviv showed a photo of the four experimental configurations, Rik Farrow interrupted to ask about the classification of each swipe pattern; Aviv responded that everything in the photo was a 4, but the projector was not rendering the photo faithfully. He also mentioned that the classifier is allowed to change the contrast of the photo in software. The experiment found that putting the phone up to the face caused a large smudge, and then entering the pattern cleaned the phone. Thus, facial contact yielded the highest retrieval rate, whereas light touches had the worst, although 37% of such photos gave at least some information about the pattern. Aviv displayed a photo illustrating that directionality of the swipe is visible because of swipe overlays at the corners; one can see which direction is on top.

The second experiment dealt with two types of simulated application usage prior to the swipe: dots due to presses of numbers or other taps, and streaks due to swipes. The worst case was when the phone is touched everywhere. Aviv also pointed out that recovery is much better when the pattern is entered after application usage instead of before, as one might expect. The third experiment dealt with two incidental clothing contact situations, both of which degraded or lost directionality information while not completely occluding the swipe.

Aviv closed by considering further work, including research into the human tendency to choose passwords with low entropy. He observed that because of the small password space, a small amount of partial information, such as a dictionary and a smudge, might be able to reduce the space below the 20-guess threshold. For example, removing passwords that include a hard-to-enter 30-degree stroke (e.g., from 1 to 8 in the standard 3x3 telephone keypad layout) reduces the pattern space by 50%.

Aviv's presentation inspired many questions. Someone asked why both the Nexus One and the G1 were included, to which Aviv responded that one screen is glass and the other is plastic. A second audience member said that it seems obvious that smudges might leave password swipe information, and asked whether there are any other phone applications where users might leave information. Aviv responded that the iPhone PIN is somewhat similar. He admitted that the iPhone on-screen keyboard is too small for smudge attacks, but speculated that iPads might be vulnerable. He closed by saying that it is difficult to determine the order of keystrokes from a screen full of on-screen keyboard smudges, unlike residual hot spots on a keypad. A third questioner asked whether people post pictures of their Android phones on Flickr. Aviv responded that one person posted a picture of a phone on a blog and asked if his pattern was discernible, and added a disclaimer that the authors were not the first to think of this attack, but they were the first to perform a systematic study. The questioner clarified that he was interested in accidental phone posts, to which Aviv replied that not many people take pictures of their phones. A fourth questioner asked how thoroughly phones had to be wiped to remove smudges. Aviv's reply was that it is fairly hard and that he found that two wipes were often necessary, and clarified that the focus of the study was whether a random picture would be able to view smudges.

Another audience member asked whether application developers could require the user to enter a random sequence to generate a random smudge as a mitigation. Aviv replied that such a solution might work, but it's putting the burden of fixing a bad security design on the user. He also said that the paper's reviewers asked for solutions to the problem, but he did not have any good solutions. He suggested numbering the dots and changing their order so as to change the pattern, but that would add 30-degree swipes. Another audience member suggested using a smaller keypad and shifting its location on the screen, but Aviv said that refocusing the camera would counter that defense. Scott Wolchok asked about the extent to which guessing the password is the easiest way to gain access to a phone, as opposed to exploiting some software vulnerability or developer access. Aviv replied that such an exploit was outside the scope of the authors' work, and noted that smudge attacks were applicable in scenarios other than finding a lost phone: an attacker might be surveilling a target, notice that the target's phone was smudged, and quickly steal the phone to recover information before replacing it. Aviv was then asked whether different screen covers (matte or glossy) mattered; he responded that dark screens would be better for security.

- ***Framing Attacks on Smart Phones and Dumb Routers: Tap-jacking and Geo-localization Attacks***
  Gustav Rydstedt, Baptiste Gourdin, Elie Bursztein, and Dan Boneh, Stanford University

Baptiste Gourdin spoke about attacks on mobile phone Web browsers. He highlighted key differences between mobile browsers and traditional browsers: the attacker can zoom to the element of his choice and easily remove browser chrome by scrolling the page down. Gourdin included a demonstration that used JavaScript to scroll down and remove the true chrome while displaying a spoofed chrome, including an SSL security indicator.

Next, Gourdin discussed tapjacking attacks. He began with a demonstration clickjacking attack on Twitter that overlaid the permanent account deletion page on top of the play button for the "BEST GAME EVER." He briefly discussed mitigations such as frame busting, but said that frame busting can crash or fail on mobile browsers. Moreover, click-

jacking protection is even rarer on mobile sites, so tapjacking amounts to "clickjacking on steroids." Gourdin provided a mobile version of his demonstration attack on Twitter and said that the vulnerability had been fixed, but was previously a live mobile Twitter vulnerability.

In the second part of his talk, Gourdin presented applications of frame leak attacks for stealing private data. He began with Paul Stone's scrolling attack from Black Hat, which allows the attacker to violate the same-origin policy and determine whether an anchor is present in a page, by placing a hashtag of the form #foo at the end of a framed URL and testing the frame's scroll position. He demonstrated how the attack could be used on Yahoo Mail Mobile to determine whether a victim received mail from a particular sender. He also pointed out that Facebook's clickjacking defense, a large dark div overlaid over the page, does not prevent frame leak attacks. Thus, an attacker can test whether a user is logged in by searching for the registration form, and can also determine which user is logged in. Facebook fixed this vulnerability by simply displaying a Facebook logo when framed, rather than showing information behind a div.

Bill Cheswick asked if the iPhone's button (used to quit the browser) mitigates these attacks. Gourdin replied that it would certainly quit the browser, but the user would still be attacked whenever he visited attacker.com.

AFTER YOU GET EIP

Summarized by Scott Wolchok (swolchok@umich.edu)

- ***Interpreter Exploitation***
  Dionysus Blazakis, Independent Security Evaluators

Dionysus Blazakis said he would show why exploit mitigations are only a safety net and vendors still need to remove bugs. From an academic point of view, he provided an example of a non-trivial information leak and showed why the leak is an emerging class of bugs. He urged academics to attempt to formalize how to find such bugs.

Blazakis discussed data execution prevention (DEP) and address space layout randomization (ASLR) and how they complicate attacks. The combination of the two makes attacks difficult, because DEP allows return-oriented programming and return-to-libc attacks, but ASLR makes such already difficult attacks probabilistic at best. An attacker looking to circumvent this combination might use information leaks and heap spraying in order to obtain executable pages with known or easily guessable locations. Blazakis then introduced two techniques, pointer inference and JIT spraying, that can be used to bypass existing exploit mitigations. The pointer inference technique interacts with the object structure of the Tamarin VM used by Flash to generate native code, in which the least significant bits of values (called "atoms") are used to encode type information. Objects are stored as tagged pointers, but integers and other primitive types are stored by value. Tamarin's general-purpose hashtable maps atoms to atoms and can be iterated over in hash order. Blazakis's insight is that the table uses the values themselves as the hash, so mixing integers and objects in the table results in integers being compared to pointers, which leaks address bits. In particular, he stated that one can determine whether an address is even or odd by putting it into two tables filled with even and odd integers and determining in which table the pointer doesn't collide. Someone asked how many bits were leaked, and Blazakis responded that about 25 bits of a 32-bit pointer could be recovered. However, the information leak is just some arbitrary heap address; there are controllable fields, but the leak is not directly exploitable.

Blazakis then moved on to JIT spraying, his second attack. Rik Farrow pointed out that JITs write code to the heap, and the pages with code have to be marked executable. Blazakis continued by explaining that a long XOR expression in ActionScript will cause the JIT to generate a compact x86 instruction stream consisting of MOV and XOR instructions, and stage-0 shellcode consisting of 2-byte instructions can be encoded into the constants manipulated in the expression. The emitted function can also contain a pointer to a string constant used to host stage-1 shellcode. Generating many such functions will effectively spray the ActionScript heap with shellcode. Blazakis demonstrated his exploit, which took about a minute.

Someone asked if these attacks meant that he had to eschew JIT programs to remain secure. Blazakis responded that in short, the answer was yes.

- ***A Framework for Automated Architecture-Independent Gadget Search***
  Thomas Dullien and Tim Kornau, zynamics GmbH; Ralf-Philipp Weinmann, University of Luxembourg

Tim Kornau spoke about the goal of using return-oriented programming tools across multiple platforms. He enumerated the common architectures today and stated that exploits should run even on a refrigerator. Specifically, the authors' goals are to execute code in the presence of the NX bit and when binaries are signed, but circumventing ASLR is outside the scope of the talk. The strategy the authors adopted was to reuse application code (i.e., through return-oriented programming) without relying on returns or return-like instructions; rather, they intend to extract semantic information from the binary. Kornau then introduced REIL, a 17-instruction RISC instruction set where all instructions are three operands and have no side effects. REIL is currently unable to support exceptions, floating-point instructions, or 64-bit computing, but those capabilities are under development. Someone asked why exception support was important; Kornau responded that, for example, MIPS's integer instructions use exceptions to represent various things, and it's difficult to model exceptions architecture-independently.

Kornau then explained the algorithms developed by the authors. In the first stage, data is collected from the binary by first extracting REIL expression trees from the native instructions and then extracting path information by bottom-up depth-limited search from the end of the gadget. All paths are stored in the same expression tree by multiplying the condition bit together with the operations. In the second stage, the expression trees for single native instructions are combined along paths and simplified (e.g., by constant folding). In the third stage, the authors locate useful gadgets by using a tree match handler determining whether a condition is met for each needed operation. The algorithm selects only the simplest gadget for each operation. Kornau stated that the algorithms are currently functional, but searching for gadgets is highly platform- and compiler-dependent. He cited difficulties like branch delay slots (MIPS), predicated execution (ARM), and register windows (SPARC). Further work includes an abstract gadget description language, an automatic gadget compiler, more platforms for REIL, and better understanding of the implications of different compilers.

Rik Farrow clarified that by "gadget" Kornau meant "a block of code that does something." Kornau replied that yes, traditionally, it does something useful and must be chainable to other gadgets. He stated that the authors' analysis differs from the traditional return-oriented programming analysis because it does not reason about unintended instructions and requires a valid disassembly up front. In reply to a second question, Kornau stated that fuzzy tree matching only searches for certain operands, because REIL has a very normal structure. A third audience member asked how large binaries had to be in order to find Turing-complete gadget sets. Kornau replied that it was very binary-dependent; he cited libsystemb as an example that generated over 240,000 gadgets and said that an attack can usually be adapted to such large binaries, whether or not the gadget set is Turing-complete. The questioner then asked how large the files were in bytes. Kornau said that he believed that libsystemb is about 200KB, but he was not certain.

- ***English Shellcode*** (Invited Talk)
  Joshua Mason and Sam Small, Johns Hopkins University; Fabian Monrose, University of North Carolina at Chapel Hill; Greg MacManus, iSIGHT Partners

Sam Small discussed how English shellcode can be used to avoid network intrusion detection systems (NIDS). He began with a review of shellcode and evading filtering and detection; shellcode transformations have been used previously to bypass application-level input filters, but, arguably, not to evade detection. He stated that NIDS works by using either regular expressions and signatures or emulation. The problem with emulation is that the attacker can use domain-specific knowledge of the application, such as registers or memory, and eflags in particular are almost always reliable. Thus, NIDS can't be aware of which paths are actually taken in a particular string, and the attacker can set eflags using arithmetic operations if necessary. Moreover, the attacker can use self-modifying code, even if he is constrained to English.

Small then moved on to the details of English shellcode generation. English shellcode has three parts: the pre-decoder, the decoder, and the transformed shellcode. The decoder unpacks the transformed shellcode, but cannot be written in English (because of instructions like lods and jnz), so the English pre-decoder is included to unpack the decoder. Small stated that the decoder would not be explained in the talk and moved on to the details of the generation engine. The language generator is based on beam search and uses a large corpus of text to build a language model. It looks at every word in the corpus that could follow the current word in the shellcode, concatenates it with the current shellcode string, and, using a scoring engine, determines how well the modified string accomplishes the desired code. The engine is in two parts: the sentinel breaks the shellcode into chunks of instructions and passes them to the executor, which it monitors through ptrace. The sentinel eventually returns a score. Small stated that the proof-of-concept system took about 12 hours, but combining the sentinel and executor into one process through a "feat of engineering" reduced the time to 20–30 minutes.

Small closed by showing some samples of English shellcode, including two quite long encodings of exit(0). The sample text, while not entirely "readable," contained many coherent phrases and popular topics. Small pointed out that the letter "r" is a jump and can be used to skip more English-like blocks of the shellcode paragraph.

Someone asked what the average size increase of English shellcode was. Small responded that there are several factors, but it is easily over 100x, which isn't prohibitive if shellcode can be placed on the heap. He said that the size increase depends on several tunable parameters that have not yet been tuned for space. Someone else asked whether the generated shellcode was contextual, as Small mentioned at the start of the presentation. Small replied that it sometimes was, and could avoid choosing instructions that access memory and registers with unknown values. A third audience member suggested that an online game could be used to get people to write sensible text to fill in the shellcode, and Small mentioned that his co-author would often ask for words that fit certain constraints during development. Someone else asked about searching for code in standard texts, such as help files. Small replied that such searching is theoretically possible, but it seems very difficult. A fifth questioner asked how much of the pre-decoder was predictable, and Joshua Mason replied that it is specifically designed to make prediction impossible. If a NIDS matched on the necessary bytes, it would also block valid text.