reflections on witty

Analyzing the Attacker

by Nicholas Weaver

Nicholas Weaver is a postdoctoral researcher at the International Computer Science Institute in Berkeley. His research focuses on modeling existing and future Internet-scale attacks, and automatic defense systems.

nweaver@icsi.berkeley.edu

and Dan Ellis

Dan Ellis is a Ph.D. student at George Mason University and a researcher at MITRE. His interests are information security and intrusion detection, and malicious code in particular.

ellisd@mitre.org

Copyright 2004 The MITRE Corporation and the International Computer Science Institute. Used with permission.

On March 20, 2004, an attacker released a single-packet UDP worm, Witty into the wild. Although only infecting roughly 12,000 machines, and less than 700 bytes long, this worm represents a dangerous trend in malicious code. The attack is well understood: There have been several analyses[1] of the worm itself, and an excellent analysis by Shannon and Moore on the network propagation,[2] including the presence of seeding or hitlisting (starting the worm on a group of systems to speed the initial propagation). But what can we learn about the attacker?

Examining the timeline of events, the worm itself, its malicious payload, and the skills required all point to an author who was motivated, sophisticated, skilled, and malicious. Although there have been previous well-engineered worms (notably the Morris worm and Nimda), Witty represents a dangerous new trend, combining both skill and malice.

It's actually unfortunate that Witty hasn't gotten the amount of attention lavished on previous worms, as it was a very significant attack. This worm contained a payload malicious to the host computer and was released with almost no time to patch systems. The worm contained no significant bugs and was written by a malicious author deeply familiar with the theoretical and practical state of the art in worm construction and computer security.

The Timeline and the Witty Worm

On March 8, eEye security discovered a stack overflow in the BlackICE/RealSecure products of ISS (Internet Security Systems). These end-host IDS products don't just restrict access based on port numbers and sender addresses like a conventional personal firewall, but include analyzers that evaluate traffic for particular applications, looking for anomalies or known vulnerabilities. One such module, the analyzer for ICQ instant messages, contained a series of stack overflows.

These vulnerabilities were relatively easy to exploit. The ICQ analyzer is triggered whenever the host computer receives a UDP packet whose source port is 4000, regardless of the destination port or the presence of a listener. Since there was no use of StackGuard,[3] /GS,[4] nonexecutable stacks, or other protections, a single-packet UDP-based overflow simply overwrites the return address, pointing to the injected code contained within the packet.

Stack overflows that can be exploited with a single UDP packet are among the most dangerous vulnerabilities, as they naturally support worms like Slammer.[5] Turning a normal single-packet stack overflow exploit into a Slammer-class worm requires a small amount of additional logic. Instead of shellcode, the attack initializes a random-number generator, gets a pointer to the start of the overflowed buffer, and goes into an infinite loop to send the packet to random addresses.

eEye quickly notified ISS, which released a patched version of the RealSecure and BlackICE products on March 9. eEye then published their advisory on March 18,[6] giving a high-level description of the vulnerability. On the evening of March 19, the Witty worm was released into the wild, less than 48 hours after eEye's public disclosure.

1. LURHQ Threat Intelligence Group, "Witty Worm Analysis," *http://www.lurhq.com/witty.html*; Kostya Kortchinsky, "Witty Worm Disassembly," *http://www.caida.org/analysis/security/witty/BlackIceWorm.html*.

2. Colleen Shannon and David Moore, "The Spread of the Witty Worm," *http://www.caida.org/analysis/security/witty/*.

3. Crispin Cowan et al., "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," *Proceedings of the 7th USENIX Security Conference*, *http://www.usenix.org/publications/library/proceedings/sec98/cowan.html*, January 1998, pp. 63–78.

4. Brandon Bray, "Compiler Security Checks in Depth," *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dv_vstechart/html/vctchcompilersecuritychecksindepth.asp*.

5. David Moore et al., "Inside the Slammer Worm," *IEEE Magazine of Security and Privacy*, July/August 2003, pp. 33–39.

6. eEye, "Internet Security Systems PAM ICQ Server Response Processing Vulnerability," *http://www.eeye.com/html/Research/Advisories/AD20040318.html*.

Witty (named for the portion in the stack overflow stating "Insert witty comment here") was an architecturally simple worm. It began by cleaning up from the buffer overflow, including obtaining API pointers from the overflowed DLL. Because of this reliance on magic numbers, Witty could only infect systems running version 3.6.16 of iss-pam1.dll, although it might crash older versions.

After cleaning up the stack overflow, the worm's code executed the main-body routines. The worm first seeded the pRNG (pseudo-random number generator) with gettickcount(), a recording of the number of milliseconds since the machine reset. It then sent out 20,000 copies of itself to random addresses with random destination ports. As an additional obfuscation, it also randomly padded the packet size. After completing this loop, it attempted to open a random physical disk and, if successful, to overwrite a random 64KB block of data. Finally, it jumped back to the seeding of the pRNG, repeating the process until the host machine finally crashed.

Although simple, the execution was superb. There were no significant bugs, especially in the pRNG (a common error in previous worms). Reseeding the pRNG each time through the main loop may have been accidental, but it was a useful flourish, papering over many possible flaws in the pRNG. The author also avoided common mistakes by using the system-provided pRNG instead of coding his own pseudo-random generator. The malicious payload, slowly corrupting the drive, causes immediate damage but does not significantly slow the worm's spread, while randomizing the destination port makes the worm more likely to penetrate firewalls.

Finally, the author seeded the worm. Rather than just starting at a single location, the worm started out on over 110 different victims.[7] Although previously a theoretical technique,[8] this represents the first occurrence of significant seeding in an autonomous worm. However, this was almost superfluous. Because Witty's single-packet nature is naturally fast, an unseeded Witty would have still spread worldwide in under two hours.

Analyzing the Attacker

Although it may not seem like much information, we can actually develop several insights into the worm author or authors. Not only was the author a skilled programmer, he (she, or they) was familiar with the lore, motivated, and malicious. He avoided the common mistakes and had access to a small, geographically distributed network of compromised systems.

The attack demonstrated considerable skill and knowledge. The attacker understood how to program in x86 assembly language and access Windows API functions. He was able to implement a stack-overflow attack. As importantly, the attacker understood worm-lore: Not only did the attacker seed the worm, he constructed a payload that was malicious to the host yet did not slow the worm's spread. Due to the short time frame, the attacker most likely knew all this information in advance rather than learning on the fly.

The attacker wrote compact code. Witty's body consists of just 177 x86 instructions in 474 bytes (the rest of the worm is the buffer overflow and padding).[9] In this small space, the author constructed routines to clean up from the overflow attack, seed the random-number generator, propagate the worm, and execute the malicious payload, demonstrating the attacker's skill at writing x86 assembly.

7. Shannon and Moore, "The Spread of the Witty Worm."

8. Stuart Staniford, Vern Paxson, and Nicholas Weaver, "How to 0wn the Internet in Your Spare Time," *Proceedings of the 11th USENIX Security Symposium*, USENIX, August 2002, *http://www.usenix.org/publications/library/proceedings/sec02/staniford.html*.

9. Kortchinsky, "Witty Worm Disassembly."

The lack of major bugs suggests that the attacker tested the worm before release. Although it would be possible for someone to write a worm without errors, it seems more likely that the worm was tested. The testing would only require a couple of systems, if the attacker monitored the network traffic on the test systems and knew the common errors made by worm authors.

But the most substantial implications arise from the short time between disclosure and the worm's release. Either the attacker discovered the vulnerability independently, reverse-engineered the patch, obtained an advance copy of the disclosure, developed the bulk of the worm in advance, or simply worked quickly.

The first option is that the attacker had developed his exploit independently of eEye's disclosure. In that case, why did he wait to release the worm? Why not release the worm after the disclosure, rather than just after a patch was released to block his exploit? Although this is still a possibility, the timing of the worm's release argues against the attacker independently discovering the ISS security flaw.

The second possibility, reverse engineering the patch before the vulnerability was disclosed, also seems unlikely. The version release notes for BlackICE (*http://blackice.iss.net/update_center/readme_pcp.txt, version 3.6.ccg*) did not mention a vulnerability fix among the changes. If the attacker was using or developed an automatic analysis tool, he might have noticed the changes and used them as a guide to creating an exploit, but he had no public indication of a vulnerability.

The third option has the attacker knowing the vulnerability before public disclosure, giving him more time to work. Thus, rather than 48 hours, the attacker would have had over a week to develop and test his code. In this case, the number of possible suspects is considerably smaller, as such information requires that the author be an insider or someone who has compromised ISS's or eEye's communication systems.

The attacker could have constructed the framework in advance, waiting for single-packet vulnerabilities to insert into his worm. If the attacker developed a generic framework, this implies that he was both malicious and premeditated, but wasn't targeting ISS or ISS users. But if the attacker was opportunistic, why use a payload that damages the host? Given the bulk of a worm, created in advance, there are many more attractive payloads (such as deploying a control network) which a generic attacker might employ. Such payloads would grant the attacker control of all the victims instead of simply corrupting them.

The final possibility is that the attacker simply worked fast. For an attacker with the required skills, it is definitely plausible that he constructed, tested, and released Witty in under two days, although the time window is relatively short.

The attacker apparently used a network of compromised machines obtained using a different mechanism, rather than a hit list of probable victims, to seed his worm. This vulnerability resists scanning (unless actually exploited, all systems react the same way), making the creation of a list of probable victims difficult. The very short timeline between disclosure and release, with no reported scanning before the worm's release, further suggests that a list of targets wasn't created in advance. Finally, the initial machines did not stop scanning in the same manner as later infections, suggesting that they were running a program to propagate the worm rather than the worm itself.

The use of previously compromised machines requires that the attacker either obtained access to 110 machines using a different tool, already had access to 110 machines, or took control of these machines from a third party. Thus Witty's author probably possessed some ties to the attacker underground, to gain access to these machines in the short time frame.

The use of a directly malicious payload also suggests the attacker had a motive. One possibility is that he was experimenting with malicious payloads. Yet since the attacker probably tested the worm, he could have verified that the payload wouldn't restrict propagation. Thus, although Witty's author could have been conducting an experiment, it seems unlikely.

Another possibility is that the attacker was targeting a particular ISS customer or group of customers without caring for collateral damage, or that the attack was an attempt to blind ISS sensors from a different, targeted attack.

The final possibility was that Witty was a direct attack against ISS by deliberately damaging ISS's customer base, or an opportunistic attack on ISS simply because ISS is one of many security companies. How many users will have second thoughts about purchasing ISS's systems? How many customers will change to a competitor? Especially when the vulnerability exploited was a stack overflow, the simplest and most easily prevented C programming error.

Witty represents a new generation of malcode: written by a motivated, skilled, and malicious individual.

Conclusion

Witty represents a new generation of malcode: written by a motivated, skilled, and malicious individual. Witty's author is the first to combine both skill and substantial malice. Witty's author had some motive that led him (or her or them) to desire a destructive effect. And Witty was written by an expert who, unless caught, could do it again.

ACKNOWLEDGMENTS

This paper was a product of group effort. Thanks to Stuart Staniford, Vern Paxson, Colleen Shannon, David Moore, Stefan Savage, Scott Tenaglia, and Jack Aiken for fruitful and interesting discussions. Nicholas Weaver receives support from NSF/DHS grant NRT-0335290. Dan Ellis receives support from the Active Worm Detection & Response MITRE Sponsored Research Project.