

Psychological Security Traps

Peiter “Mudge” Zatko

DURING MY CAREER OF ATTACKING SOFTWARE AND THE FACILITIES THEY POWER, many colleagues have remarked that I have a somewhat nonstandard approach. I tended to be surprised to hear this, as the approach seemed logical and straightforward to me. In contrast, I felt that academic approaches were too abstract to realize wide success in real-world applications. These more conventional disciplines were taking an almost completely random tack with no focus or, on the opposite end of the spectrum, spending hundreds of hours reverse-engineering and tracing applications to (hopefully) discover their vulnerabilities before they were exploited out in the field.

Now, please do not take this the wrong way. I’m not condemning the aforementioned techniques. In fact I agree they are critical tools in the art of vulnerability discovery and exploitation. However, I believe in applying some shortcuts and alternative views to envelope, enhance, and—sometimes—bypass these approaches.

In this chapter I’ll talk about some of these alternative views and how they can help us get inside the mind of the developer whose code or system we engage as security professionals.

Why might you want to get inside the mind of the developer? There are many reasons, but for this chapter we will focus on various constraints that are imposed on the creation of code and the people who write it. These issues often result in suboptimal systems from the security viewpoint, and by understanding some of the environmental, psychological, and philosophical frameworks in which the coding is done, we can shine a spotlight on which areas of a system

are more likely to contain vulnerabilities that attackers can exploit. Where appropriate, I'll share anecdotes to provide examples of the mindset issue at hand.

My focus for the past several years has been on large-scale environments such as major corporations, government agencies and their various enclaves, and even nation states. While many of the elements are applicable to smaller environments, and even to individuals, I like to show the issues in larger terms to offer a broader social picture. Of course, painting with such a broad brush requires generalizations, and you may be able to find instances that contradict the examples. I won't cite counterexamples, given the short space allotted to the chapter.

The goal here is not to highlight particular technologies, but rather to talk about some environmental and psychological situations that caused weak security to come into being. It is important to consider the external influences and restrictions placed on the implementers of a technology, in order to best understand where weaknesses will logically be introduced. While this is an enjoyable mental game to play on the offensive side of the coin, it takes on new dimensions when the defenders also play the game and a) prevent errors that would otherwise lead to attacks or b) use these same techniques to game the attackers and how they operate. At this point, the security game becomes what I consider *beautiful*.

The mindsets I'll cover fall into the categories of learned helplessness and naïveté, confirmation traps, and functional fixation. This is not an exhaustive list of influencing factors in security design and implementation, but a starting point to encourage further awareness of the potential security dangers in systems that you create or depend on.

Learned Helplessness and Naïveté

Sociologists and psychologists have discovered a phenomenon in both humans and other animals that they call *learned helplessness*. It springs from repeated frustration when trying to achieve one's goals or rescue oneself from a bad situation. Ultimately, the animal subjected to this extremely destructive treatment stops trying. Even when chances to do well or escape come along, the animal remains passive and fails to take advantage of them.

To illustrate that even sophisticated and rational software engineers are subject to this debilitating flaw, I'll use an example where poor security can be traced back to the roots of backward compatibility.

Backward compatibility is a perennial problem for existing technology deployments. New technologies are discovered and need to be deployed that are incompatible with, or at the very least substantially different from, existing solutions.

At each point in a system's evolution, vendors need to determine whether they will forcibly end-of-life the existing solutions, provide a migration path, or devise a way to allow both the legacy and modern solutions to interact in perpetuity. All of these decisions have numerous ramifications from both business and technology perspectives. But the decision is usually

driven by business desires and comes down as a decree to the developers and engineers.* When this happens, the people responsible for creating the actual implementation will have the impression that the decision has already been made and that they just have to live with it. No further reevaluation or double guessing need take place.

Imagine that the decision was made to maintain compatibility with the legacy technology in its replacement. Management further decrees that no further development or support work will take place on the legacy solution, in order to encourage existing customers to migrate to the replacement.

Although such decisions place burdens on the development in many ways—with security implications—they are particularly interesting when one solution, usually the new technology, is more secure than the other. In fact, new technologies are often developed *explicitly* to meet the need for greater security—and yet the old technology must still be supported. What security problems arise in such situations?

There are different ways to achieve backward compatibility, some more secure than others. But once the developers understand that the older, less secure technology is allowed to live on, solutions that would ease the risk are often not considered at all. The focus is placed on the new technology, and the legacy technology is glued into it (or vice versa) with minimal attention to the legacy's effects. After all, the team that is implementing the new technology usually didn't develop the legacy code and the goal is to ultimately supplant the legacy solution anyway—right?

The most direct solution is to compromise the robustness and security strength of the new technology to match that of the legacy solution, in essence allowing both the modern and legacy technology to be active simultaneously. Learned helplessness enters when developers can't imagine that anything could be done—or worse, even should be done—to mitigate the vulnerabilities of the legacy code. The legacy code was forced on them, it is not perceived to be their bailiwick (even if it impacts the security of the new technology by reducing it to the level of the old), and they feel they are powerless to do anything about it anyway due to corporate decree.

A Real-Life Example: How Microsoft Enabled L0phtCrack

Years ago, to help system administrators uncover vulnerabilities, I wrote a password-cracking tool that recovered Microsoft user passwords. It was called L0phtCrack at the time, later to be renamed LC5, and then discontinued by Symantec (who had acquired the rights to it) due to concerns that it could be considered a munition under the International Tariff on Arms Regulations (ITAR).† Many articles on the Net and passages in technical books have been written about how L0phtCrack worked, but none have focused on *why* it worked in the first

* Or at least it often appears to the developers and engineers that this is the case.

† This might not be the end of L0phtCrack....

place. What were some of the potential influences that contributed to the vulnerabilities that L0phtCrack took advantage of in Microsoft Windows?

In fact, the tool directly exploited numerous problems in the implementation and use of cryptographic routines in Windows. All these problems originated in the legacy LAN Manager (or LANMAN) hash function that continued to be used in versions of Windows up to Vista. Its hash representation, although based on the already aging Data Encryption Standard (DES), contained no salt. In addition, passwords in LANMAN were case-insensitive. The function broke the 14-character or shorter password into two 7-byte values that were each encrypted against the same key and then concatenated. As I described in a post to BugTraq in the late 1990s, the basic encryption sequence, illustrated in Figure 1-1, is:

1. If the password is less than 14 characters, pad it with nulls to fill out the allocated 14-character space set aside for the password. If the password is greater than 14 characters, in contrast, it is truncated.
2. Convert the 14-character password to all uppercase and split it into two 7-character halves. It should be noted that if the original password was 7 or fewer characters, the second half will always be 7 nulls.
3. Convert each 7-byte half to an 8-byte parity DES key.
4. DES encrypt a known constant (“KGS!@#%\$”) using each of the previously mentioned keys.
5. Concatenate the two outputs to form the LM_HASH representation.

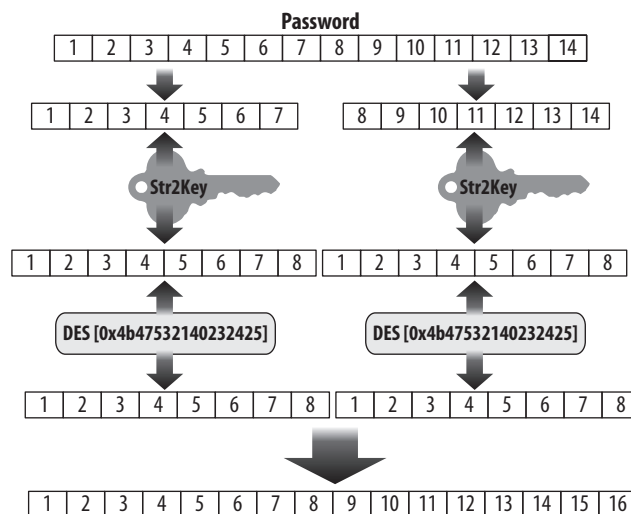


FIGURE 1-1. Summary of old LANMAN hash algorithm

This combination of choices was problematic for many technical reasons.

The developers of Windows NT were conscious of the weaknesses in the LANMAN hash and used a stronger algorithm for its storage of password credentials, referred to as the NT hash. It maintained the case of the characters, allowed passwords longer than 14 characters, and used the more modern MD4 message digest to produce its 16-byte hash.

Unfortunately, Windows systems continued to store the weaker version of each password next to the stronger one—and to send both versions over the network each time a user logged in. Across the network, both the weaker 16-byte LANMAN hash and the stronger 16-byte NT hash underwent the following process, which is represented in Figure 1-2:

1. Pad the hash with nulls to 21 bytes.
2. Break the 21-byte result into three 7-byte subcomponents.
3. Convert each 7-byte subcomponent to 8-byte parity DES keys.
4. Encrypt an 8-byte challenge, which was visibly sent across the network, using the previously mentioned DES keys.
5. Concatenate the three 8-byte outputs from step 4 to make a 24-byte representation that would be sent over the network.

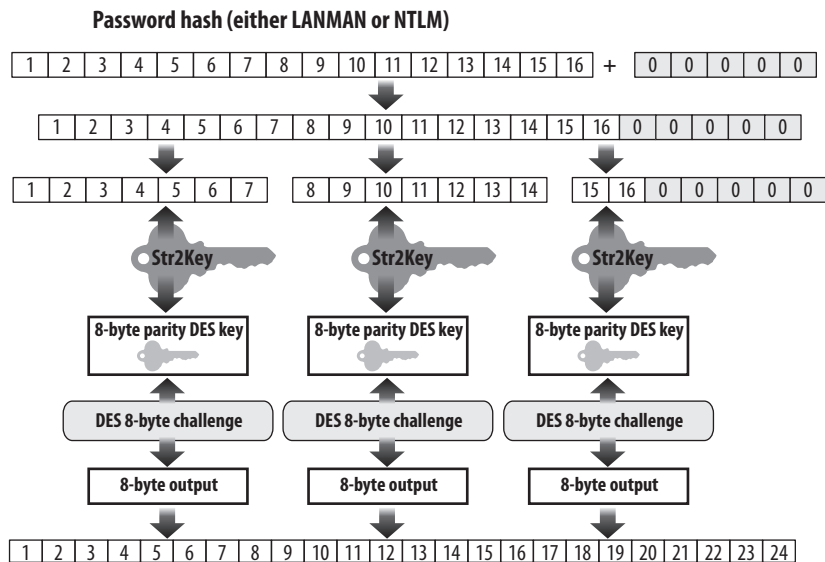


FIGURE 1-2. Handling both LANMAN and NT hashes over the network

Microsoft preferred for all their customers to upgrade to newer versions of Windows, of course, but did not dare to cut off customers using older versions or even retrofit them with the new hash function. Because the password was a key part of networking, they had to assume that, for the foreseeable future, old systems with no understanding of the new hash function would continue to connect to systems fitted out with the more secure hash.

If systems on both sides of the login were new systems with new hash functions, they could perform the actual authentication using the stronger NT hash. But a representation of the older and more vulnerable LANMAN hash was sent right alongside its stronger sibling.

By taking the path of least resistance to backward compatibility and ignoring the ramifications, Microsoft completely undermined the technical advances of its newer security technology.

L0phtCrack took advantage of the weak LANMAN password encoding and leveraged the results against the stronger NTLM representation that was stored next to it. Even if a user chose a password longer than 14 characters, the cracking of the LANMAN hash would still provide the first 14, leaving only a short remnant to guess through inference or brute force. Unlike LANMAN, the NT hash was case-sensitive. But once the weak version was broken, the case specifics of the password in the NT hash could be derived in a maximum of 2^x attempts (where x is the length of the password string) because there were at most two choices (uppercase or lowercase) for each character. Keep in mind that x was less than or equal to 14 and thus trivial to test exhaustively.

Although NTLM network authentication introduced a challenge that was supposed to act as a salt mechanism, the output still contained too much information that an attacker could see and take advantage of. Only two bytes from the original 16-byte hash made it into the third 7-byte component; the rest was known to be nulls. Similarly, only one byte—the eighth—made it from the first half of the hash into the second 7-byte component.

Think of what would happen if the original password were seven characters or less (a very likely choice for casual users). In the LANMAN hash, the second group of 7 input bytes would be all nulls, so the output hash bytes 9 through 16 would always be the same value. And this is further propagated through the NTLM algorithm. At the very least, it takes little effort to determine whether the last 8 bytes of a 24-byte NTLM authentication response were from a password that was less than eight characters.

In short, the problems of the new modern security solution sprang from the weaker LANMAN password of the legacy system and thus reduced the entire security profile to its lowest common denominator. It wasn't until much later, and after much negative security publicity, that Microsoft introduced the capability of sending only one hash or the other, and not both by default—and even later that they stopped storing both LANMAN and NT hashes in proximity to each other on local systems.

Password and Authentication Security Could Have Been Better from the Start

My L0phtCrack story was meant to highlight a common security problem. There are many reasons to support multiple security implementations, even when one is known to be stronger than the others, but in many cases, as discussed earlier, the reason is to support backward compatibility. Once support for legacy systems is deemed essential, one can expect to see a fair amount of redundancy in protocols and services.

The issue from a security standpoint becomes how to accomplish this backward compatibility without degrading the security of the new systems. Microsoft's naïve solution embodied pretty much the worst of all possibilities: it stored the insecure hash together with the more secure one, and for the benefit of the attacker it transmitted the representations of both hashes over the network, even when not needed!

Remember that *learned helplessness* is the situation where one comes to the conclusion that he is helpless or has no recourse by training rather than from an actual analysis of the situation at hand. In other words, someone tells you that you are helpless and you believe them based on nothing more than their "say so." In engineering work, learned helplessness can be induced by statements from apparent positions of authority, lazy acceptance of backward compatibility (or legacy customer demand), and through cost or funding pressures (perceived or real).

Microsoft believed the legacy systems were important enough to preclude stranding these systems. In doing this they made the decision to keep supporting the LM hash.

But they took a second critical step and chose to deal with the protocol problem of legacy and modern interactions by forcing their new systems to talk to both the current protocol and the legacy one without considering the legacy security issues. Instead, they could have required the legacy systems to patch the handful of functions required to support logins as a final end-of-life upgrade to the legacy systems. Perhaps this solution was rejected because it might set a dangerous precedent of supporting systems that they had claimed had reached their end-of-life. They similarly could have chosen not to send both old and new hashes across the network when both systems could speak the more modern and stronger variant. This would have helped their flagship "New Technology" offering in both actual and perceived security.

Ultimately Microsoft enabled their systems to refrain from transmitting the weaker LANMAN hash representation, due to persistent media and customer complaints about the security weakness, in part prompted by the emergence of attack tools such as L0phtCrack. This shows that the vendor could have chosen a different path to start with and could have enabled the end users to configure the systems to their own security requirements. Instead, they seem to have fallen victim to the belief that when legacy support is required, one must simply graft it onto the new product and allow all systems to negotiate down to the lowest common denominator. This is an example of learned helplessness from the designer and implementer standpoints within a vendor.

NOT MICROSOFT ALONE

Lest the reader think I'm picking on Microsoft, I offer the following equal-opportunity (and potentially offending) observations.

During this time frame (the mid- to late 1990s), Microsoft was taking the stance in marketing and media that its systems were more secure than Unix. The majority of the servers on the Internet were Unix systems, and Microsoft was trying to break into this market. It was well known that numerous security vulnerabilities had been found in the various Unix variants that made up the vast majority of systems on the public Internet. Little research, however, had been performed on the security of Microsoft's Windows NT 4.0 from an Internet perspective. This was due in no small part to the fact that NT 4.0 systems were such a small fraction of the systems on the Net.

Microsoft's stance was, in essence, "we are secure because we are not Unix." But it took until the Vista release of the Windows operating system for Microsoft to really show strong and modern security practices in an initial OS offering. Vista has had its own issues, but less on the security front than other factors. So, when NT 4.0 was novel, Microsoft picked on Unix, citing their long list of security issues at the time. The shoe went on the other foot, and people now cite the litany of Microsoft security issues to date. Now that Microsoft actually offers an operating system with many strong security components, will there be anyone to pick on? Enter Apple.

Apple Computer has seemingly taken a similar marketing tack as Microsoft did historically. Whereas Microsoft essentially claimed that they were secure because they were not Unix, Apple's marketing and user base is stating that its OS X platform is more resistant to attacks and viruses essentially because it is not Windows. Having had a good look around the kernel and userland space of OS X, I can say that there are many security vulnerabilities (both remote and local) still waiting to be pointed out and patched. Apple appears to be in a honeymoon period similar to Microsoft's first NT offering: Apple is less targeted because it has a relatively tiny market share. But both its market share and, predictably, the amount of attack focus it receives seems to be increasing....

Naïveté As the Client Counterpart to Learned Helplessness

As we've seen, the poor security choice made by Microsoft in backward compatibility might have involved a despondent view (justified or not) of their customers' environment, technical abilities, and willingness to change. I attribute another, even larger, security breach in our current networks to a combination of learned helplessness on the part of the vendor and naïveté on the part of the customer. A long trail of audits has made it clear that major manufacturers of network switches have intentionally designed their switches to "fail open" rather than closed. Switches are designed to move packets between systems at the data-link layer. *Failing closed*, in this case, means that a device shuts down and stops functioning or otherwise ceases operation in a "secure" fashion. This would result in data no longer passing

through the system in question. Conversely, *failing open* implies that the system stops performing any intelligent functions and just blindly forwards all packets it receives out of all of its ports.‡

In essence, a switch that fails open turns itself into a dumb hub. If you're out to passively sniff network traffic that is not intended for you, a dumb hub is just what you want. A properly functioning switch will attempt to send traffic only to the appropriate destinations.

Many organizations assume that passive network sniffing is not a viable threat because they are running switches. But it is entirely common nowadays to connect a sniffer to a switched LAN and see data that is not destined for you—often to the extreme surprise of the networking group at that organization. They don't realize that the vendor has decided to avoid connectivity disruptions at all costs (probably because it correctly fears the outrage of customer sites whose communications come to a screeching halt), and therefore make its switches revert to a dumb broadcast mode in the event that a switch becomes confused through a bug, a security attack, or the lack of explicit instructions on what to do with certain packets. The vendor, in other words, has quietly made a decision about what is best for its customer.

I would like to believe that the customer would be in a better position to determine what is and what is not in her best interest. While it might be a good idea for a switch to fail open rather than shut down an assembly line, there are situations where switches are used to separate important traffic and segregate internal domains and systems. In such cases it might be in the best interest of the customer if the switch fails closed and sends an alarm. The customer should at least be provided a choice.

Here we have both learned helplessness on the vendor's part and naïveté on the consumer's part. The learned helplessness comes from the vendor's cynicism about its ability to educate the customer and get the customer to appreciate the value of having a choice. This is somewhat similar to the previous discussion of legacy system compatibility solutions. The vendor believes that providing extra configurability of this kind will just confuse the customer, cause the customer to shoot herself in the foot, or generate costly support calls to the vendor.

The naïveté of the client is understandable: she has bought spiffy-looking systems from well-established vendors and at the moment everything seems to be running fine. But the reasonableness of such naïveté doesn't reduce its usefulness to an adversary. *Must a system's security be reduced by an attempt to have it always work in any environment?* Are protocols blinded deliberately to allow legacy version systems to interact at weaker security levels? If a system gets confused, will it revert to acting as a dumb legacy device? These situations can often be traced back to learned helplessness.

‡ This is the opposite of electrical circuits, where failing closed allows current to flow and failing open breaks the circuit.

Confirmation Traps

Hobbit (friend and hacker[§] extraordinaire) and I were having dinner one August (back around 1997) with an executive and a senior engineer^{||} from Microsoft. They wanted to know how we were able to find so many flaws in Microsoft products so readily. I believe, with imperfect memory, that we replied truthfully and said we would input random garbage into the systems. This is a straightforward bug- and security-testing technique, sometimes known as “fuzzing,” and has now been documented in major computer science publications. However, fuzzing was not widely practiced by the “hacker” community at the time.

We told the engineer we were surprised how often Windows would go belly-up when confronted with fuzzed input. We followed up by asking what sort of robustness testing they performed, as it would seem that proper QA would include bad input testing and should have identified many of the system and application crashes we were finding.

The engineer’s response was that they performed exhaustive usability testing on all of their products, but that this did not include *trying* to crash the products. This response shone a light on the problem. While Microsoft made efforts to ensure a good user experience, they were not considering adversarial users or environments.

As an example, teams that developed Microsoft Word would test their file parsers against various acceptable input formats (Word, Word Perfect, RTF, plain text, etc.). They would not test variations of the expected formats that could be created by hand but could never be generated by a compatible word processor. But a malicious attacker will test these systems with malformed versions of the expected formats, as well as quasi-random garbage.

When we asked the senior Microsoft representatives at dinner why they did not send malicious data or provide malformed files as input to their product’s testing, the answer was, “Why would a user want to do that?” Their faces bore looks of shock and dismay that anyone would intentionally interact with a piece of software in such a way as to intentionally try to make it fail.

They never considered that their applications would be deployed in a hostile environment. And this view of a benign world probably sprang from another psychological trait that malicious attackers can exploit: *confirmation traps*.

An Introduction to the Concept

Microsoft’s product testing was designed to *confirm* their beliefs about how their software behaved rather than *refute* those beliefs. Software architects and engineers frequently suffer

[§] The word “hacker” is being used in the truest and most positive sense here.

^{||} Sometimes it seems it is cheaper to hire a key inventor of a protocol and have him “reinvent” it rather than license the technology. One of the people responsible for Microsoft’s “reimplementation” of DCE/RPC into SMB/CIFS was the engineer present at the dinner.

from this blind spot. In a 1968 paper, Peter Wason pointed out that “obtaining the correct solution necessitates a willingness to attempt to falsify the hypothesis, and thus test the intuitive ideas that so often carry the feeling of certitude.”# He demonstrated confirmation traps through a simple mental test.

Find some people and inform them that you are conducting a little experiment. You will provide the participant with a list of integers conforming to a rule that he is supposed to guess. To determine the rule, he should propose some more data points, and you will tell him whether each of his sets of points conform to the unspoken rule. When the participant thinks he knows what the rule is, he can propose it.

In actuality, the rule is simply *any three ascending numbers*, but you will keep this to yourself. The initial data points you will provide are the numbers 2, 4, and 6.

At this point, one of the participants might offer the numbers 8, 10, and 12. You should inform her that 8, 10, 12 does indeed conform to the rule. Another participant might suggest 1, 3, and 5. Again, you would confirm that the series 1, 3, and 5 conforms to the rule.

People see the initial series of numbers 2, 4, and 6 and note an obvious relationship: that each number is incremented by two to form the next number. They incorporate this requirement—which is entirely in their own minds, not part of your secret rule—into their attempts to provide matching numbers, and when these sequences conform, the confirmation pushes them further down the path of confirming their preconceived belief rather than attempting to refute it.

Imagine the secret rule now as a software rule for accepting input, and imagine that the participants in your experiment are software testers who believe all users will enter sequences incremented by two. They won't test other sequences, such as 1, 14, and 9,087 (not to mention -55, -30, and 0). And the resulting system is almost certain to accept untested inputs, only to break down.

Why do confirmation traps work? The fact is that we all like to be correct rather than incorrect. While rigid logic would dictate trying to test our hypotheses—that all inputs must be even numbers, or must be incremented by two—by proposing a series that does not conform to our hypothesis (such as 10, 9, 8), it is simply human nature to attempt to reinforce our beliefs rather than to contradict them.

“Does a piece of software work as expected?” should be tested not just by using it the way you intend, but also through bizarre, malicious, and random uses. But internal software testing rarely re-creates the actual environments and inputs to which software will be subjected, by regular end users and hostile adversaries alike.

“Reasoning About a Rule,” Peter Wason, *The Quarterly Journal of Experimental Psychology*, Vol. 20, No. 3. 1968.

The Analyst Confirmation Trap

Consider an intelligence analyst working at a three-letter agency. The analyst wants to create valid and useful reports in order to progress up the career ladder. The analyst culls information from multiple sources, including the previous reports of analysts in her position. The analyst then presents these reports to her superior. While this might seem straightforward, it entails a potential confirmation trap. Before her superiors were in the position to review her work, it is quite likely that *they* were the prior analysts that created some of the reports the current analyst used as background. In other words, it is not uncommon that the input to a decision was created by the people reviewing that decision.

It should be apparent that the analyst has a proclivity to corroborate the reports that were put together by her boss rather than to attempt to challenge them. She might fall into line quite consciously, particularly if she is trying to make a career in that community or organization, or do it unconsciously as in Wason's example with three ascending numbers. At the very least, the structure and information base of the agency creates a strong potential for a self-reinforcing feedback loop.

I have personally witnessed two cases where people became cognizant of confirmation traps and actively worked to ensure that they did not perpetuate them. Not surprisingly, both cases involved the same people that brought the intelligence analyst scenario to my attention and who confirmed my suspicions regarding how commonly this error is made in intelligence reports.

Stale Threat Modeling

During a previous presidency, I acted as an advisor to a key group of people in the Executive Office. One of my important tasks was to express an opinion about a briefing someone had received about cyber capabilities (both offensive and defensive) and which areas of research in those briefings were valid or had promise. I would often have to point out that the initial briefings were woefully inaccurate in their modeling of adversaries and technologies. The technology, tactics, and capabilities being presented were not even close to representative of the techniques that could be mustered by a well-financed and highly motivated adversary. Many of the techniques and tactics described as being available only to competent nation-state adversaries were currently run-of-the-mill activities for script kiddies and hobbyists of the day.

The briefings did try to understand how cyber threats were evolving, but did so unimaginatively by extrapolating from historical technology. Technology had progressed but the models had not, and had been left far behind reality. So the briefings ended up regurgitating scenarios that were possibly based in accurate generalizations at one point in the past, but were now obsolete and inaccurate. This is endemic of confirmation traps. And as it turned out, the briefings I had been asked to comment on had come about due to situations similar to the aforementioned analyst confirmation trap.

Rationalizing Away Capabilities

As the success of the L0pht in breaking security and releasing such tools as L0phtCrack became well known, the government developed a disturbing interest in our team and wanted to understand what we were capable of. I reluctantly extended an invitation to a group from the White House to visit and get a briefing. Now, mind you, the L0pht guys were not very comfortable having a bunch of spooks and government representatives visiting, but eventually I and another member were able to convince everyone to let the “govvies” come to our “secret” location.

At the end of the night, after a meeting and a dinner together, we walked the government delegation out to the parking lot and said our goodbyes. We watched them as they walked toward their cars, concerned to make sure all of them actually drove away. So our paranoia spiked as we saw them stop and chat with each other.

I briskly walked over to the huddle and interrupted them with an objection along the lines of: “You can’t do that! You can tell all the secrets you want once you are back in your offices, but we just let you into our house and extended a lot of trust and faith in doing so. So I want to know what it is you were just talking about!” It’s amazing that a little bit of alcohol can provide enough courage to do this, given the people we were dealing with. Or perhaps I just didn’t know any better at the time.

I think this stunned them a bit. Everyone in their group of about five high-level staff looked at one member who had not, up to that point, stood out in our minds as the senior person (nice operational security on their part). He gazed directly back at me and said, “We were just talking about what you have managed to put together here.”

“What do you mean?” I pressed.

He replied, “All of the briefings we have received state that the sort of setup with the capabilities you have here is not possible without nation-state-type funding.” I responded that it was obvious from what we had showed them that we had done it without any money (it should be noted that it is a great oversight to underestimate the capabilities of inquisitive people who are broke). “We were further wondering,” he said, “if any governments have approached you or attempted to ‘hire’ you.” So in my typical fashion I responded, “No. Well, at least not that I’m aware of. But if you’d like to be the first, we’re willing to entertain offers....”

Even with this poor attempt at humor, we ended up getting along.

But despite the fear on both sides and the communication problems that resulted from our radically different viewpoints, the government team left understanding that our exploits had truly been achieved by a group of hobbyists with spare time and almost no money.

The visitors were the people who received reports and briefings from various three-letter agencies. They were aware of how the career ladder at these agencies could be conducive to confirmation biases. Assured by officials that our achievements required funding on a scale that could only be achieved by specific classes of adversaries, they took the bold step of

searching us out so that they might refute some of the basic beliefs they had been taught. They went so far as to visit the dingy L0pht and ended up modifying their incorrect assumptions about how much effort an adversary might really need to pull off some pretty terrifying cyber-acts.

Unfortunately, there are not as many people as one might like who are either able or willing to seek out uncomfortable evidence to challenge assumptions. When testing software and systems, it is important to consider the environment in which engineers, developers, and testers might be working and the preconceived notions they might bring. This is particularly important in regards to what their application might be asked to do or what input might be intentionally or unexpectedly thrust at them.

Functional Fixation

Functional fixation is the inability to see uses for something beyond the use commonly presented for it. This is similar to the notion of first impressions—that the first spin applied to initial information disclosure (e.g., a biased title in a newspaper report or a presentation of a case by a prosecutor) often permanently influences the listener’s ongoing perception of the information.

When someone mentions a “hammer,” one normally first thinks of a utilitarian tool for construction. Few people think first of a hammer as an offensive weapon. Similarly, a flame-thrower elicits images of a military weapon and only later, if at all, might one think of it as a tool to fight wildfires through prescribed burning tactics that prevent fires from spreading.

Functional fixation goes beyond an understanding of the most common or “default” use of a tool. We call it fixation when it leaves one thinking that one knows the *only* possible use of the tool.

Consider a simple quarter that you find among loose change in your pocket. If someone asks you how to use it, your first response is probably that the coin is used as a medium of exchange. But, of course, people use coins in many other ways:

- A decision-maker
- A screwdriver
- A projectile
- A shim to keep a door open
- An aesthetic and historic collectible

Ignoring these alternative functions can surprise you in many ways, ranging from offers to buy your old coins to a thunk in the head after you give a quarter to a young child.

Vulnerability in Place of Security

Now that you have a general understanding of functional fixation, you might be wondering how it relates to computer and network security.

Many people think of security products such as vulnerability scanners and anti-virus software as tools that increase the security of a system or organization. But if this is the only view you hold, you are suffering from functional fixation. Each of these technologies can be very complex and consist of thousands of lines of code. Introducing them into an environment also introduces a strong possibility of new vulnerabilities and attack surfaces.

As an example, during the early years of vulnerability scanners, I would set up a few special systems on the internal networks of the company that I worked for. These systems were malicious servers designed to exploit client-side vulnerabilities in the most popular vulnerability scanners at the time. Little did I realize that client-side exploitation would become such a common occurrence in malware infection years later.

As one example, the ISS scanner would connect to the finger service on a remote system to collect remote system information. However, the scanning software had a classic problem in one of its security tests: the program did not check the length of the returned information and blindly copied it into a fixed-size buffer. This resulted in a garden-variety buffer overflow on the program's stack. Knowing this about the scanner, and knowing the architecture of the system the scanner was running on, I set up malicious servers to exploit this opportunity.

When the company I was employed by would receive their annual audit, as a part of evaluation the auditors would run network vulnerability scans from laptops they brought in and connected to the internal network. When the scanner would eventually stumble across one of my malicious servers, the scanning system itself would be compromised through vulnerabilities in the scanning software.

This often resulted in humorous situations, as it gave the executives of the company some ammunition in responding to the auditors. Since the compromised auditor system had usually been used for engagements across multiple clients, we could confront them with audit information for other companies that were now exposed by the auditors' systems. The executives could justifiably claim that vulnerabilities found on our internal systems (living behind firewalls and other defensive technologies) were not as severe a risk to the corporation as disclosure of sensitive information to competitors by the auditors themselves—made possible by the "security software" they used. Functional fixation might cause one to forget to check the security of the security-checking software itself.

Modern anti-virus software, unfortunately, has been found to include all sorts of common programming vulnerabilities, such as local buffer overflows, unchecked execution capabilities, and lack of authentication in auto-update activities. This security software, therefore, can also become the opening for attackers rather than the defense it was intended for.

The preceding examples are straightforward examples of functional fixation and can be attributed to the same naïveté I discussed in the section on learned helplessness. However, there are more subtle examples as well.

Sunk Costs Versus Future Profits: An ISP Example

One of the greatest hampers to security springs from negative perceptions of security requirements at a high corporate level. Some of these represent functional fixation.

Several months before the historic Distributed Denial of Service (DDoS) attacks that temporarily shut down major service providers and commercial entities (including eBay, CNN, Yahoo!, and others) on the Internet,* I had the opportunity to analyze backbone router configurations for a Tier 1 ISP. The majority of the IP traffic that transited these core routers was TCP traffic, in particular HTTP communications. A much smaller percentage was UDP, and well below that, ICMP. I was surprised to discover that the routers lacked any controls on traffic other than minimal filters to prevent some forms of unauthorized access to the routers themselves. But when I suggested that the core router configurations be modified toward the end of protecting the ISP's customers, the expression of surprise shifted to the company's executives, who immediately told me that this was not an option.

Two schools of thought clashed here. The ISP did not want to risk reducing the throughput of their core routers, which would happen if they put any type of nontrivial packet filtering in place. After all, an ISP is in the business of selling bandwidth, which customers see as throughput. Router behavior and resulting throughput can be negatively impacted when the systems moving packets from point A to point B have to spend any extra time making decisions about how to handle each packet.

Furthermore, neither the ISP nor its customers were suffering any adverse effects at the time. The managers could understand that there might be an attack against their own routers, but were willing to wait and deal with it when it happened. To spend money when there was no problem might be wasteful, and they would probably not have to spend any more money on a future problem than they would have to spend now to proactively keep the problem from happening. Attacks on customers were not their problem.

On my side, in contrast, although there had not been a widespread instance of DDoS at this point in time (in fact, the phrase DDoS had yet to be coined), I was aware of the possibility of network resource starvation attacks against not only the ISP's routers but also the customers behind them. I knew that attacks on customers would be hard to diagnose and difficult to react to quickly, but I entirely failed to convince the ISP. In fact, I had to concede that from a business standpoint, their reasons for not wanting to further secure their systems was somewhat logical.

* "Clinton fights hackers, with a hacker," CNN, February 15, 2000 (<http://web.archive.org/web/20070915152644/http://archives.cnn.com/2000/TECH/computing/02/15/hacker.security/>).

(The problem of security as a cost rather than a revenue generator is also examined in Chapter 12, *Oh No, Here Come the Infosecurity Lawyers!*, by Randy V. Sabett.)

Some time after the wide-scale DDoS attacks, I was honored to find myself sitting at the round table in the Oval Office of the White House only a few seats down from President Clinton. The meeting had been called to discuss how government and industry had handled the recent DDoS situation and what should be done going forward.

And once again, I was surprised. The main concern expressed by executives from the commercial sector was that the attacks might prompt the government to come in and regulate their industry. They seemed uninterested in actually understanding or addressing the technical problem at hand.

Then it started to dawn on me that the ISPs were functionally fixated on the notion that government intervention in these sorts of matters is likely to negatively impact revenue. This was the same fixation that I had witnessed when interacting with the large ISPs months earlier in regards to placing packet filters on their core routers: that security costs money and only prevents against future potential damage. They never considered ways that implementing security could *create* revenue.

After the meeting, I reengaged the executive of the large ISP I had previously dealt with. I told him that I understood why he made the security decisions he had and asked him to give me an honest answer to a question that had been on my mind lately. I asked him to suppose I had not approached him from a security standpoint. Instead, suppose I had pointed out that the ISP could negotiate committed access rates, use them to enforce caps on particular types of traffic at particular rates, take these new certainties to better plan utilization, and ultimately serve more customers per critical router. Further, they could use such a scheme to provide different billing and reporting capabilities for new types of services they could sell. The filtering and measurement would prevent inappropriate bandwidth utilization by the client, but any useful traffic the client found to be blocked or slowed down could be satisfied by negotiating a different service level.

But as a side effect, the same filtering would dramatically reduce inappropriate bandwidth utilization by external acts of malice. Would this, I asked, have been a better approach?

The answer was a resounding *yes*, because the company would view this as an opportunity to realize more revenue rather than just as an operational expense associated with security posturing.

I learned from this that I—along with the vast majority of practitioners in my field—suffered from the functional fixation that security was its own entity and could not be viewed as a by-product of a different goal. As so often proves to be the case, architecting for efficiency and well-defined requirements can result in enhanced security as well.

Sunk Costs Versus Future Profits: An Energy Example

Part of my career has involved examining in detail the backend control systems at various electric utilities and, to a somewhat lesser extent, oil company backend systems. I assessed how they were protected and traced their interconnections to other systems and networks. It was surprising how the oil and electric industries, while using such similar systems and protocols, could be operated and run in such widely disparate configurations and security postures.

To put it politely, the electric company networks were a mess. Plant control systems and networks could be reached from the public Internet. General-purpose systems were being shared by multiple tasks, interleaving word processing and other routine work with critical functions that should have been relegated to specialized systems to prevent potential interference or disruption of operations. It appeared in several cases that systems and networks had been put together on a whim and without consideration of optimal or even accurate operations. Implementers moved on to the next job as soon as things worked at all. Many plant control networks, plant information networks, and corporate LANs had no firewalls or chokepoints. From a security standpoint, all this combined to create the potential for malicious interlopers to wreak serious havoc, including manipulating or disrupting the physical components used in the production and transmission of power.

Conversely, the few offshore oil systems that I had looked at, while utilizing similar SCADA systems, were configured and operated in a different fashion. Plant control and information networks were strictly segregated from the corporate LAN. Most critical systems were set correctly to have their results and status handled by a librarian system that then pushed the information out in a diode fashion to higher analysis systems. Concise and efficient network diagrams resulted in crisp and clean implementations of SCADA and DCS systems in the physical world, including restriction of access that resulted in effective security. In many cases the components were custom systems designed and configured to perform only specific functions.[†]

The contrast between the electric and oil organizations intrigued and worried me. As fate would have it, I was in the position to be able to call a meeting about this subject with some high-ranking technical people from electric companies, oil companies, and government (think spook) agencies.

The first salient aspect that surprised me from the meeting was that the people from the electric utilities and their electric utility oversight and clearinghouse organizations did not refute my statements regarding the poor—or completely missing—security on their networks and systems. This surprised me because the electric companies were publicly denying that they had

[†] It is important to note that I analyzed only a subset of all the oil and electric systems out there. The differences are put forth here for comparison purposes to help illustrate functional fixation and how it affects corporate views of *security*. The oil industry has its fair share of incorrectly configured systems and environments, as do almost all large industries. Similarly, there are probably some well-configured electric company plant information and control networks...somewhere.

any cyber-system risk. In our meeting they pointed out some examples where security had been implemented correctly—but they acknowledged that these examples were exceptions and not the norm.

My second surprise came when the oil companies stated that they did not go about designing their systems from a security perspective at all, and that although security was important, it was not the business driver for how things were configured. The primary driver was to have an edge against their direct competitors.

If company A could make a critical component operate at 5% greater efficiency than company B, the increased operational capacity or reduction in overhead rewarded company A over time with large sums of money. Examples of how to increase such efficiency included:

- Forced separation and segregation of systems to prevent critical systems from incurring added latency from being queried by management and reporting requests
- Utilizing special-purpose systems designed to accomplish specific tasks in place of general-purpose nonoptimized systems

These efficiencies benefited security as well. The first created strong, clean, and enforceable boundaries in networks and systems. The second produced systems with smaller surface areas to attack.

Enforceable network and system boundaries are an obvious effect, but the case of smaller surface areas deserves a brief examination. Imagine that you have a general-purpose system in its default configuration. The default configuration might have several services already configured and running, as well as many local daemons executing to assist user processing. This allows the system to be deployed in the largest number of settings with minimal reconfiguration required. The systems' vendor prefers such systems with broad capabilities because they make installation easier.

However, this doesn't mean that the default setting is *optimal* for the majority of consumers, just that it is *acceptable*. In the default setting, each of the running services is an attack surface that may be exploited. Similarly, client applications may be compromised through malicious input from compromised or falsified servers. The more services and client applications that are running on the system, the greater the attack surface and the greater the likelihood that the system can be remotely or locally compromised.

Having a large attack surface is not a good thing, but the drawback of generality examined by the oil companies was the systems' suboptimal performance. For each running program, which includes server services as well as local applications, the kernel and CPU devotes processing time. If there are many running applications, the system has to time-slice among them, a kernel activity that in itself eats up resources.

However, if there are few running applications, each one can have a greater number of CPU slices and achieve greater performance. A simple way to slim down the system is to remove superfluous services and applications and optimize the systems to run in the most

stripped-down and dedicated fashion possible. Another way is to deploy systems dedicated to specific functions without even the capability of running unrelated routines. These tactics had been used by the oil companies in the offshore rigs I had examined in order to maximize performance and thus profits.

Why hadn't the electric utilities gone through the same exercise as the oil companies? At first, electric companies were regulated monopolies. Where these companies did not need to be competitive, they had no drive to design optimized and strictly structured environments.

One would be tempted to assume that deregulation and exposure of electric companies to a competitive environment would improve their efficiency and (following the same path as oil companies) their security. However, the opposite occurred. When the electric companies were turned loose, so to speak, and realized they needed cost-cutting measures to be competitive, their first steps were to reduce workforce. They ended up assigning fewer people to maintain and work on the same number of local and remote systems (often through remote access technologies), focusing on day-to-day operations rather than looking ahead to long-term needs. This is usually a poor recipe for efficiency or security.

The story of the oil companies confirms the observation I made in the previous section about the ISP. Most organizations think of security as a sunk cost, insofar as they think of it at all. Security approached in this fashion will likely be inadequate or worse. If, however, one focuses on optimizing and streamlining the functionality of the networks and systems for specific business purposes, security can often be realized as a by-product. And once again, security professionals can further their cause by overcoming their functional fixation on security as a noble goal unto itself worth spending large sums on, and instead sometimes looking at sneaking security in as a fortuitous by-product.

Summary

In this chapter, I have offered examples of classic security failures and traced them beyond tools, practices, and individual decisions to fundamental principles of how we think. We can improve security by applying our resources in smarter ways that go against our natural inclinations:

- We can overcome learned helplessness and naïveté by ensuring that initial decisions do not shut off creative thinking.
- We can overcome confirmation traps by seeking inputs from diverse populations and forcing ourselves to try to refute assumptions.
- We can overcome functional fixation by looking for alternative uses for our tools, as well as alternative paths to achieve our goals.

All these ventures require practice. But opportunities to practice them come up every day. If more people work at them, this approach, which I'm so often told is unusual, will become less curious to others.