Security in the Real World: How to Evaluate Security Technology

By Bruce Schneier

The following remarks are excerpted from a general session presentation delivered at CSI's NetSec Conference in St. Louis, MO, on June 15th, 1999.

At Counterpane Systems, we evaluate security products and systems for a living. We do a lot of breaking of things for manufacturers and other clients. Over the years, I've built a body of lore about the ways things tend to fail. I want to share my "top 20 list" of what's wrong with security products these days.

Cryptography is a really neat technology, because it allows us to take existing business and social constructs from the real world and move them into the world of computer networks. This is actually the big idea of cryptography. It doesn't do anything new, it doesn't do anything magical. All it does is take existing things and move them onto networks.

Privacy, fairness, authentication—we know how they work and what to do with them. But now we want to do them on networks; we want to do them remotely. Cryptography has the potential of transforming the Internet, or any network, from an academic toy into a real business tool by allowing us to do real business—for example, signing and enforcing contracts or doing e-commerce.

Unfortunately, most of the products I see aren't very good. They have problems, they're broken. Most cryptography in these products doesn't perform as advertised, and neither do some monitors.

Programming Satan's Computer

Security engineering is not like any other type of engineering. An engineer who's building something will spend all night to make it work. That's quintessentially what a good hack is. It works, it's functional. In a normal product, it's what it does that's impressive.

But security products are not useful because of what they do; they're useful precisely because of what they don't allow to happen. Security has nothing to do with functionality.

If you were to build a word processor and wanted to know if it printed, you could plug a printer in, push the print button, and see if a printed document came out. If you're building a encryption product, you can put a file in, watch it encrypt and decrypt. You know it works, but you have no idea if it's secure or not. And that's a big deal. What it means is that you can't tell if a product's secure simply by examining it, simply by running it through functional tests.

No amount of beta testing will find a security flaw. In many ways, security engineering is similar to safety engineering. But there is a difference. Safety engineering has to do with making something work in the presence of random or transient faults (i.e., Murphy's Law). Security programming involves making sure something works even in the presence of a malicious adversary who will make exactly the wrong thing fail at exactly the wrong time and do it again, and again, and again, and again to break the security. That's why I call it programming Satan's computer. You program a computer with the assumption that a malicious adversary intent on defeating the system is living inside the system. Security is supposed to provide some way to encapsulate him.

Testing Satan's Computer

Security is orthogonal to functionality. Security has nothing to with what the product does, or

how well it does it, or how good the user interface is. You can't give a product to a thousand random people, have them beta test it for a month and really learn anything about the security. They can tell you if it works and how functional it is, but they can't tell you if it's broken or not. Generally, to test security, at least in the real world, you just put the product out there and experienced security professionals, either working for industry, or in academia, or working on their own (commonly known as hackers), find flaws and alert the *New York Times* and you get your feedback that way. Not terribly useful. But it's where we've ended up.

Failure of Testing Security

Imagine a vendor shipping a product without any functional testing. No in-house testing, no beta testing. The developers just create the product, make short compiles, and then ship it. It's inconceivable that a product built that way would not have any bugs. It's hard enough to get rid of all the bugs when you have all of those testings. But without them, it's just not going to happen.

Now, let's look at a security product. Imagine a security product shipping without any security testing. Similarly, the odds of it shipping without any security flaws is negligible. And—this is an important point—very few people do real security testing. They do testing, but not security testing.

How to Test Security

Experienced security testers can discover security flaws. It's not easy, it's not fast, but it can be done.

Security is like a chain; the weakest link will break it. You can find security flaws in any part of the product—for example, design, algorithms, protocols, implementation, configuration, user interface, procedures, how it's installed, how it interacts with other products.

Black box testing is something I never do, although I'm asked to regularly. In black box testing, you're given the product in a shipped form and asked to break it. It isn't a terribly useful test. You can do that and lots of flaws are found that way, but it's not a real good usage of your testing dollar, because a lot of the time is going to be spent reverse engineering the C code, or figuring out how it works, or doing something else that you can just mirror by giving someone the actual product, rather than analyzing the security. But when we do good security tests, we get everything. We get the source code, we get the design documents, we get to see how it works, and then we look for security flaws that way. That's a much more cost-effective way of evaluating products. That way, we can find things and fix them.

Flaws can be anywhere: the threat model, design, algorithms, protocols, implementation, configuration, user interface, usage procedures, etc. "Black box" testing isn't very useful. There is no comprehensive security checklist. Experience in real-world failures is the only way to be a good tester.

Unfortunately, there's no comprehensive security checklist. There isn't a list of a hundred things that I can refer to and say, "they all check, the product is secure."

Envision your house. How do you know if your house is secure? Are the doors secure? Yes. Are the windows secure? Yes. Does that mean your house is secure? Maybe. Where's the key to the house?

A group of art thieves in California would break into people's houses by cutting holes in their wall with a chainsaw. That's a really interesting attack against a house. It bypasses most security measures. So whether your house is secure is not an easy thing to determine. And it's based on who the attacker is, what's inside

the house, what kind of threats you face. You can't just run through a checklist and say, "Yes, the house is secure."

Why Cryptosystems Fail

Why do cryptosystems fail? Why do things break in the real world? Well, the reasons are as numerous as there are systems. Every time I put this list together I'm always amazed by some product vendor that invents a new way. But there are a lot of common mistakes. Unfortunately, we've seen a lot of them since the 1970s. And probably since even before if we bothered to look. I don't mean this list to be exhaustive. Vendors should feel free to make new mistakes; they're always entertaining to see.

I'm going from the inside out, from the mathematical out to the systems—that's the basic roadmap. They're not in entertaining order, so this isn't actually a "Top Ten" list. And they're not in frequency. They're really from the math out to the user.

Use of Proprietary Algorithms

The first one is very mathematical. A lot of products these days still use proprietary algorithms. And this, to me, is amazing.

We have any number of known and trusted (to various degrees) cryptographic algorithms in the literature, designed by academics, reviewed by academics, and available for free. Source code is available. It's no work to use them and yet companies again and again use proprietary algorithms. This is bad, because proprietary algorithms are rarely any good.

Designing cryptographic algorithms is very, very difficult.

If you've been watching the Advanced Encryption Standard (AES) process that's wending its way through the NIST approval process, you know that 15 algorithms were submitted from a worldwide call for algorithms meeting a certain criteria. Four of them have been broken already. One of those four was broken during the question session when the algorithm was presented.

Designing crypto algorithms is very difficult. Many published algorithms are insecure. Almost all unpublished algorithms are insecure. Unless someone has considerable experience crytpanalyzing algorithms, it is unlikely that his design will be secure. It is easy for him to create an algorithm that he himself cannot break.

The submission document for the algorithm I submitted with my colleagues at Counterpane was the length of a book. This is hard to do. But even normally rational people tend to be blinded by a bright shiny new algorithm. It seems so easy. The unfortunate truth is anybody can design an algorithm that he himself cannot break. It's actually profound. Anyone out there, from the best cryptographer to the random person on the street, can sit down with a pencil and paper, design an algorithm and say, "I can't break it." And then here's the fallacy—because you can't break it, you make the following assumption: "Therefore, it must be secure."

So we end up with lots of proprietary algorithms. We have a lot of amateur cryptanalysts who will design the algorithms, do some work and then say, "Look I can't break it, therefore it's secure." My feeling is that if the designers haven't proven themselves by breaking several published algorithms, why should I look at their designs? The odds of them being secure are pretty negligible. Indeed, the top five AES candidates—this is top five based on a formal poll of cryptographers—were actually de-

signed by teams that have cryptanalysts on them. They seem to be the fastest, the most elegant, the best performing, the ones that seem to be the most secure. Still, nobody trusts them—give us a couple of years to stare at them. Eventually, we're going to have a new standard.

There's no reason that I can think of ever to use a new and unanalyzed algorithm. There's never any benefit. There might be the personal pride of the designer. Other than that, you might as well use a known algorithm. So the moral there is "never, ever trust a proprietary algorithm."

The design process of public proposal, analysis, revision, repeat seems to work pretty well. Compare the security of the proprietary Microsoft PPTP with open IPSEC. There is no substitute for peer review. A closed or proprietary protocol is most likely flawed.

Use of Proprietary Protocols

Similarly, we have lots of proprietary protocols. And again, it's the same lesson. Designing protocols is very, very hard. Many published protocols have been broken—years after their publication flaws were discovered. People who design protocols tend to be a specialized breed and sort of an odd lot. They've built all sorts of interesting tools to do protocol analysis. There are some great tools for automated protocol checking. They're basically reasoning engines that look at a protocol and decide if you can get knowledge that you're not allowed to. But even so, many companies will not use standards—whether the standard is Kerberos or IPSec or SSL. But they'll develop their own thing. This, to me, is also odd. The public process of, "Does it propose, analyze, revise, etc.?" is a nice process. We've seen this with a number of protocols—for example, with S/MIME for e-mail, with IPSec for Internet. It's actually a really good process. You end up, after several years, with a well-designed and robust protocol. That's a good process. Compare this with the Microsoft PPTP protocol, which was designed by Microsoft, in their own labs for their own use—it turned out to be pretty badly broken. And I published a paper on that last year. They have since released a fix, which is better but is still broken.

Again, there's no substitute for peer review. One person can't do it. The public process is a good process. And given that all of these protocols are free, it's kind of silly to invent your own. You don't have the same expertise as the public community does. You might as well let us do your work for you, that's what we're here for. (Actually it's not, but we like to pretend we are.)

A closed protocol is most likely flawed. Consider the protocols that are being used for digital cell phones, a whole bunch of broken protocols, and broken algorithms too. That's another example of an industry that said, "Let's do it ourselves, let's not engage the community, let's have closed proprietary standards." You're going to see the same thing with digital content, the DVD and DIVX protocols, they're going to be broken within the year.

Bad Randomization

This is actually an interesting problem. It is also probably the most serious problem in products that use cryptography. Random numbers are critical for cryptography. You need them for keys, seeds for generating keys, random values to make protocols work, and random values to make digital signatures work. If your random number is broken, it doesn't matter how good your algorithm protocol is. An insecure random number generator (RNG) can

compromise the entire system. One of the flaws of Netscape Navigator that made the press was a break in the random number generator. It didn't matter how good the algorithm was, the keys that came into it were very bad. Lots and lots of products fail this way. It's hard to do.

There are a bunch of ways to abuse random number generators, not just the obvious ones of looking at them and seeing that they're weaker than you think. You can learn the state at a time and predict that state at a future time. You can control some of the inputs. I once looked at an old version of a cryptographic token, which had a good random number generator—except that when you plugged it into a certain reader you could manipulate it so the input stopped coming in and it would tail down to a known state. An interesting way to break a system. We fixed that one.

There are many ways to abuse RNGs: learn the state, extend a state compromise forward or backward in time, learn or control inputs to reduce output entropy. Poor RNGs are probably the most common security problem in products today. Counterpane Systems has released Yarrow, a public domain RNG. Use it.

There aren't any good standards for random number generators. There are some guidelines, but they are not strong against real-world attacks. And the guidelines that exist only talk about generating the random numbers—hardware noise sources and the like—and not about how to use them. The attacks we look at don't break the random number generators at the analog source, but the method of using the random numbers after they turn into bits. So yes, there's that clever hack where they threw a camera at a bunch of lava lamps. That makes really good random input, but our analysis starts after that, after the analog lava lamps are converted into a digital stream and are then used by the application.

To help alleviate this problem, I released a public domain random number generator called Yarrow, which is secure as far as I know against all the attacks I know of. And companies are starting to use it. It's free, the source code's available. I urge companies to use it. Hopefully, this will improve the general state of random numbers in the world, which can only help.

Cult of Mathematics

You see this problem with a lot of vendors. There's a one time pad, therefore it's secure. There's a proof, therefore it's secure. I tend to be very frustrated at these sorts of things because as a mathematician I wish it were true. Mathematics is a science, it's not a measure of security.

Another example of this is the cult of key length. It has a 500-bit key, therefore it's better than a 300-bit key. We used 5,000-bit RSA, therefore we're better than 2,000-bit RSA. It's not actually true, or at least it's not true in any meaningful sense.

I guess it was last year that IBM released the system that was going to take over cryptography and make us all safe. Well, we haven't seen it yet. They had a protocol, there was a security proof, but all it means is that, given some certain assumptions, the protocol is secure against certain attacks. Lots of systems have been broken despite proofs. There are two parts to a proof—there are the assumptions, then the logical steps of the proof. If I am going to try to analyze a system that has a proof and the proof is valid, I'm not going to attack the logic, that's kind of silly. I'm going to attack the assumptions. We see this again and again and again. Remember the attack against PKCS#1 from Bell Labs—a really nice attack that attacked the assumptions of how you pack bits into a digital signature? There are

lots of one time pad systems that I've broken. You don't break the one time pad, you break the way it's used. You can go to the NSA Web site and read about the Venona project, where they broke one time pads from the Soviet Union. They were one time pads, they were just used improperly.

Mathematics works on bits, but security involves people. I'll often say to potential clients, "If you give me bits, I can secure them. It's that human–bit interface that is going to cause you hardship. That's where things are going to break."

Proofs of security only works within the model of the security proof. For example, the attack on PKCS #1 went outside the mathematical model of RSA to break certain implementation. Mathematics works on bits; security involves people.

Insecure Software, Computers and Networks

Insecure software, computers, networks—I sort of lump them all together. You can look at all the CERT advisories. You'll see the same mistakes being made over and over again. The moral is that a buzzword-compliant product isn't enough. Just because the product literature reads, "We use triple DES, we use RSA," doesn't mean it's secure. These algorithms might be necessary for security, but they're certainly not sufficient.

It's really difficult to write secure code. That's something we've learned. We're starting to build tools to help us. There are tools that automatically check for overflow bugs or other security holes. Unfortunately, they're not widely used. They're actually not easy to use, either.

There are user interface errors. I've seen products in which the window where you enter your key will break the product because the window helpfully saves the last thing entered somewhere. That's a really interesting break because it happens nowhere near the security portion of the code. Again, the moral is you have to look at everything. Pieces of code far removed from the security can affect security.

Similarly, it's impossible to build a secure application on top of an insecure operating system. You can make your buildings secure, but if the foundation that you've built it on is wobbly, it's not going to do you any good.

Let's assume that PGP is a perfect product, and there's no way to break it. You still can, on the computer it's running from, install a stealth keyboard recorder, which will capture keystrokes and capture your passphrase. It has nothing to do with PGP, but will break you completely.

You can install a Trojan horse that might act like PGP, but actually ships copies of the keys somewhere. Source code is available, you can write that kind of program. I've seen it done in academic demonstrations. You can install a virus that does the same thing.

PGP could be perfect, but there will still be ways in.

Similarly, an insecure network can undermine the security of a computer it's attached to. Windows NT has C2 security only if it's standalone. Attach it to a network and suddenly vulnerabilities go up.

This the chain of security. Often the weak links are the software, computers, and networks. And they're links you might not have any control over. Things you don't know about, just because you're attached to a network, can come and attack you and cause profound effects.

Failures of Secure Perimeters

Another thing you see a lot in products is the notion that there is some tamperproof element in the system that attackers can't get at. For example, we see this in smart cards, we see this in secure computers sitting in safes. These are

all examples of secure perimeters. It's kind of a neat idea. And if it works, there's a lot we as security professionals can do with it. You can sort of think of a smart card as a branch office of the bank living in your wallet. It's a trusted agent of the bank that's sitting in your wallet constantly managing your balance. Think of an electronic cash card, or the door key card for your hotel room—these are smart cards. These devices work as long as the secure perimeter works.

But the unfortunate reality is that secure perimeters are really hard to do. There's no such thing as tamperproof hardware. If you speak to people who work in the industry, in tamperproofing, they'll tell you it can't be done.

There's a bar; and you can raise the bar. You can make it cost a good amount of money to fake the tamperproof seal on a bottle of Bayer aspirin. But you can't make it impossible. And a lot of the attacks we've seen on smart cards in the past couple of years—for example, the side channel attacks done by myself and Paul Kocher of Cryptography Research Inc., the reverse engineering attacks done by Ross Anderson, and Marcus Kuhn in their lab at Cambridge University in the U.K.; the transient fault attacks by Biham and Shamir in Israel and folks at Bell Corp. There are lots of ways into this particular secure perimeter. It doesn't actually work as a secure perimeter. A lot of companies tend to rely on it anyway. I prefer, in designs, to use secure perimeters but not to rely on them. My moral is any system where the device is owned by one person and the secrets within the device are owned by another is a fundamentally flawed system.

A great analogy is a slot machine. A slot machine on a casino floor is a secure perimeter. If you can get into that slot machine you can make a lot of money. But that slot machine is sitting on a casino floor and there are guards and there are cameras and there are lights and there are people and if you go near that slot machine with a drill you're going to be carted off to jail. But if the casino said to you, "Here, take this slot machine, take it home, take it home for a month. Play it, bring it back and we'll pay whatever's on the pay line." That's a much riskier proposition. And that's basically what happens with smart cards: "Here's a smart card, take it home, take it to your lab, do what you want and we'll honor whatever balance shows up on it next month." It's dangerous.

Any system where the device is owned by one person and the secrets within the device are owned by another is an insecure system. Systems should use tamper-resistance as a barrier to entry, not as an absolute security measure.

New Vulnerabilities Introduced by Key Escrow

Another danger I like to talk about is key escrow. It's an interesting issue and there are some real good business cases made in various directions.

By key escrow, I mean key escrow, data escrow, trusted third-party encryption, data recovery, key recovery, whatever the political word for it is today. It's all the same threat model. And I'm trying to stay at that level.

Data backup is vital. Anybody who's worked in computers for more than a month knows that. You need to back up your data. And when you encrypt your data, the value of the key equals the value of the data. In a very basic sense, cryptography takes large secrets and replaces them with small secrets. So you have this huge file, or this e-mail, or whatever, that you need to keep secure. You encrypt it with this small key and now you need to keep that small key secure.

Now that data has a value, and since you need the key to get at the data, that key has that same value. There are a lot of reasons why you're going to want to back up encryp-

tion keys used to encrypt stored data because they have tremendous value.

You don't want to lose your accounts receivable database when your CFO gets run over by a bus. That would be bad for your business. On the other hand, data in transit has absolutely no value because you can always ask for it again. I didn't receive the mail, send it again. I'm talking on the phone, we get disconnected, you call back. Data in transit has no value; therefore, there is no reason to back up keys used to encrypt data in transit.

So that's the business case. The government case is separate. And the government case is also a lot more rigorous than the average business case.

Corporate data backup is based on corporate data requirements. You don't actually need 24-hour access, surreptitious access or real-time access. When you lose a file, you call the computer center and say, "I erased this file," and within an hour or two or a little bit more, you get the file back. It happens rarely, that's probably good enough.

Law enforcement requirements make it much harder for a corporation to do this well.

Adding backups to a key escrow system will make whatever security you have worse. They reduce the security.

Have you ever used a STU III or a similar secure phone? It's kind of a neat idea. It uses a public key. You pick up the phone, you call somebody. The two of you negotiate a key, and when you hang up the key disappears. So that key only exists for the lifetime of a call. If the enemy overran your company and confiscated all your equipment, they could never recover the key used for that phone call because the key has disappeared. It's created at the inception of the call and it's destroyed at the completion of the call.

That's a really good security property. We call that perfect forward secrecy. And we like to see that in systems.

A key escrow database destroys that property. Suddenly, there's a copy of the key that exists after the call is completed, and might exist for a year or two. And might be managed by minimum wage clerks typing in requests for keys from beleaguered FBI employees. So adding a key escrow system grafts this enormous infrastructure on what was a very simple system.

Law enforcement access requirements fly in the face of security requirements solved by cryptography. The massive scale of any key-escrow infrastructure is beyond the ability of the current security community. The NSA's own analysis concluded that key escrow is too risky, and creates more security problems than it solves.

So there are three problems: losing perfect forward secrecy, turning what was simple into complexity, and then adding a very large target.

What was once a distributed system of various telephones being used for different security calls has now been encapsulated in this huge target that has the keys for all of them—a much bigger risk. And to me, while I am here explaining why relatively simple systems are hard to do, I don't see that it is possible to build the large key escrow infrastructure that seems to be mandated, required, or requested. The NSA's own report says just about the same thing.

The politics of this is changing very rapidly. The U.S. seems to be alone in its demand for key escrow. France has reversed, Germany has reversed, England has reversed. We seem to be trying to do better. And hopefully, we will do even better.

Reliance on User-Remembered Secrets

This is a tough problem to solve, and it's a big one. In a lot of systems, security is based on the user-remembered secret.

PGP has excellent algorithms and protocols, but all the security hinges on the passphrase that the user types in to unlock his private key.

There are PGP cracker tools available that try to brute-force this passphrase. If you chose a lousy one, it can be broken.

The security of your computer network is based on the password. Most exploits into computer systems are based on grabbing that encrypted password file and trying to brute-force bad passwords.

Even worse, a lot of systems have the characteristic of only being as secure as the weakest password. Now this is really annoying. You might have a shared computer system with a thousand users. Your system could be as secure as the worst password among all the ones that those thousand users pick.

Many systems rely on user-generated and user-remembered secrets for security (for example, PGP). Many password-protected systems have the characteristic of being only as secure as the weakest password. Users cannot remember good secrets.

If I were to steal the password file and run L0phtcrack on it, I would get the one password that's "dog" or "cat." And it's actually even worse than that. You can run dictionaries that are millions of common names long. This is hard because users cannot remember good secrets. We really can't rely on users to remember good secrets.

English has about 1.3 bits of entropy per character. So if you were to tell the user to pick a 30-character English passphrase, that's about the same as a 40-bit key. There is uppercase, lowercase, alphanumerics, non-English characters, and punctuation. Generously, there's 4 bits of entropy per character. So a 12-character password is about 40 bits, a little bit more secure than a 40-bit key.

Now with the advances we've been seeing in brute force technology, 56-bit keys are within easy reach of anybody with a quarter million dollars to spend on a machine. This is bad.

As security professionals, we can't expect our users to type in paragraphs when they want to access their computer. We're going to get some help from the biometric community. This is a place where they can help us. There is a lot of stuff biometrics can't do; this is something it actually can.

Reliance on Intelligent Users

Similarly, a lot of systems rely on the intelligence of users, and this is also problematic. There's been a battle for the past 20 years, because security would like to push security up higher in the application stack, closer to the user, but the users don't want to see it; they want to push security down, way down into the network layer and the transport.

Just as users will take a chair and prop open a fire door in order to make their lives easier, they'll bypass security measures to make their lives easier. "Here, you can borrow my password. I'm going out for the next day."

You're surfing the Web and you see a button on the Web site saying, "Click here to see the dancing pigs." And you click on the Web site and then this window comes up saying, "Warning: this is an untrusted Java applet. It might damage your system. Do you want to continue? Yes/No."

Well, the average computer user is going to pick dancing pigs over security any day. And we can't expect them not to.

I use Netscape and I can set the security level low, medium, or high. For most people, that's setting the annoyance level. "How many annoying dialog boxes do I have to see?"

You can't trust users to make these decisions. You can't trust users to verify certificates. SSL only works if, after you establish a connection,

you go and verify the certificate that you received. Pretty much nobody ever does that. I mean, I never do it. You can't expect users to follow security policies and procedures. You really can't, we know this from the real world.

As I said before, cryptography works in the digital world. But it's very hard to manage that transition from the people into the digital world. And that's where we see a lot of breaks—social engineering breaks, user interface breaks.

Users cannot be relied upon to follow security procedures. Users will work around annoying security measures. Users will give away secrets (for example, through "social engineering"). Cryptography works in the digital world; it is very difficult to manage the transition from people into the digital world and back again.

What you see isn't necessarily what you get. If you type an e-mail message and you push the encrypt button, you have no clue what happened. You assume it encrypted it if you click the button. What did you just sign? Did you sign what was on the screen? Well, you think you did, you hope you did. You have no idea. If it's a malicious computer program, it could have made you sign something else.

This brings us into the notion of the authentication infrastructure, the public infrastructure, which as it's built today has a lot of weaknesses.

First, it doesn't actually encapsulate trust. Trust is a very complex social phenomenon. Trusting to ride in someone's car, to lend them five dollars, let them baby-sit your children, to get married to them, to go into business with them—there's a lot of different definitions of trust. We really can't encapsulate those in a single certificate. And that's really what PKI vendors are trying to do.

There's no single global namespace in this world. There's no single level of assurance. Open up your wallet and you'll see any number of analogue certificates. You'll see a driver's license, you'll see credit cards, you'll see airline frequent flyer cards, you'll see library cards, Starbucks frequent coffee drinker cards, etc. There's no reason why you can't have just one certificate in your wallet and have it be used for all those applications. There's no reason why the credit card company can't use your driver's license. There's no reason why the airline can't use your passport ID number. These are all just keys into databases. But that has never happened and it never will. Every entity needs to manage, issue, use, and revoke their own certificates based on their own rules.

That same thing is going to happen in the digital world. You're not going to have this single monolithic certificate that'll be used for everything. Businesses won't stand for it, and also it's less secure. A big monolithic security certificate manufacturer is a big target.

I remember talking with Marcus Ranum, and we decided that the VeriSign route key could be stolen for 50 million dollars, which is about how much it would cost to make a leveraged buyout of the company. There could be a situation where it's worth it.

Similarly, certificates issued by these trusted third parties don't make any sense without some liability model attached. If you're going to use VeriSign certs, what's your recourse if VeriSign has a problem? You need something. And similarly in the real world, I said this before, key verification is not done. This is the hidden secret of PKI. There's no way of checking revocation, and people don't try to check anyway. Revocation is not being done.

Authentication is not the same thing as authorization. This is important! Imagine you get an e-mail that's a purchase order for a million dollars of computer equipment. You can go through all the steps of verifying the cryptography, verifying the signature, verifying the cert that signed, the public key that signed, the pur-

chase order, verifying that it hasn't been revoked, verifying that it has a trusted path from them to you. It verifies that the message is valid. But you'd still have no answer to the real question, which is "Should I accept this purchase order?" Not "Is this key valid?" but "Is this key authorized to do what it's trying to do?" And very few systems address this harder and far more interesting problem.

What is the key allowed to do, what is the authorization associated with this key? Not what is the authentication. Authentication is actually the easy part.

Key revocation is very difficult, and it is currently not being done securely. When you get a key over the Internet, it is vital to verify that the key has not been revoked or stolen. Authentication is not the same thing as authorization. Authentication is automatic; authorization requires thought.

Reliance on Global Secrets

Many systems—generally not Internet systems, usually proprietary systems like cell phone systems or smart card systems—are created with global secrets. There is one key. If you break the key, you've compromised the system. That's bad. The security is only as strong as the weakest instantiation of that key. They're only protected to the degree of the least trusted person who has access to that secret.

Similarly, when a compromise occurs (this is not "if," this is when), it can be very hard to recover. If the same global secret is used in all of the fare collection terminals throughout a particular subway system, or in every cell phone, or in every DVD player, or in every European satellite TV decoder, when it's compromised it's very hard to recover. This is not a very good design feature.

Version Rollback Attacks

Another bad design feature we see a lot are systems that are backwards compatible with insecure systems. I call these version rollback attacks. You invent version 1.0 of your system, it's broken. You release version 2.0 but it's compatible with 1.0.

In a lot of these systems, you can build attacks where you convince the version 2.0 systems to default to the 1.0 protocol. And then you break it that way. This can be hard, because on the Internet, you want to be backwards compatible with everything. But security, by it's nature, shouldn't be backwards compatible.

Paul Kocher, in developing SSL, spent a lot of time making sure SSL 3 was backwards compatible just enough to be functional, but not enough to allow this version rollback attack. It's a very hard problem to solve and a lot of systems don't solve it.

"Below the Radar" Attacks

Automating a system helps both the defender and the attacker. Suddenly, attacks that were too cumbersome in the real world can be automated to a degree that they become profitable. Take for example, the great story (which I'm pretty sure is apocryphal) of stealing the fractions of pennies from interest-bearing accounts. That's an attack that would have made no sense in the real world. But in a computer world where you can automate it, you can actually make a lot of money.

Another such attack would be e-mailing a million people with stock predictions and then having them randomly go up, go down, and paring down your database until you get a string much smaller that has a string of ten successes. That's another attack that's much more likely because you can automate it.

In the mid-1970s, we gave or sold the Shah of Iran our old currency printing presses. The Aya-

tollah took them over when he seized power. He realized quickly that it's much more profitable to print American dollars than it was to print the local currency, and they've been doing that ever since. And that's why we have new bills. That's an example of an interesting attack because it shows the different characteristics of an attack in the computer world versus the real world. In the real world, this was a very devastating attack.

When the Secret Service went to the Treasury people and said, "Look, these bills are coming in, we can't stop them, what should we do?" The Treasury people used their calculators and concluded, "Well they have this many printing presses, printing this many bills per hour, this many bills per year but it doesn't affect the money supply. Don't worry about it. We'll fix it eventually." It's interesting. What that says is that this attack, as devastating as it is, only has a maximum amount of damage it can do.

Many systems have a "default to insecure" mode of operation. For example, crash a firewall. Wait until someone gets frustrated and shuts it down. Waltz right in. Disrupt the communications line that verifies a Visa card, and a merchant will just accept the transaction. Only the military has the discipline not to communicate if the security measures are not in place (and they're by no means perfect).

Now on the Internet, if you can imagine being able to mint electronic cash in a similar way, you can post that piece of software on a Web site; a thousand people have it in an hour, a million people have it in a week. The amount of damage it can do is would be awesome. It could drop a currency. That's interesting. That means the characteristics of propagation, because of the automation, are much different than they are in the real world.

I expect some pretty interesting exploits because people don't understand this. The moral is only the first attacker needed skill. If he automates the attack, he can send it out and script kiddies can run it.

Imagine someone coming up with a Web site that said, "Click here to bring down the Internet." Certainly, it is possible. Thankfully, we haven't seen it yet. But someone with skill doesn't necessarily have ethics.

Poor Failure Modes

Systems tend to break around the edges. Systems almost never break by hammering them in the middle; they break around the edges. One of the problems is that systems don't fail well. They default to insecure.

If you're buying something with a credit card and the clerk runs it through the VeriPhone terminal, there's online credit verification to make sure you're not using a stolen card. And if it doesn't work, let's say you've scratched out the magnetic stripe, then the merchant is going to run it manually. So the security measure fails just at the point you want it.

If someone wants to get into a network, one of the easier ways is to crash their firewall. It's much easier to do denial-of-service attacks than to break in. But if you crash it two dozen times in as many hours, eventually someone's going to complain and it'll probably go away while they try to fix it.

In the real world, we tend not to have the discipline not to communicate if the communication security isn't in place. This is a real problem I don't have a good solution for.

Poor Compromise Recovery

We don't have good procedures for compromised recovery. A lot of the products deal with prevention, but don't handle recovery very well.

To me, recovery is even more important. Be-

cause there are going to be problems; people are going to get in, you're going to lose security, you're going to lose the root key to your banking protocol. How do you recover? What do you do now? Yes, it will be expensive, yes, it will be embarrassing. But you don't want to close up shop and go home.

Improper Risk Management

It's vital to understand what it is we're protecting. Who are the attackers are? Are they criminals? Are they bored grad students? What is the value of the data? Is it very valuable, is it cheap? How long does it have to be secure?

In the brokerage world, today's secrets are tomorrow's headlines. Stock trading data has to be kept secret until the trade is executed. Some data has to be kept secret for years, some for longer. You need to understand who the attackers are, what the value of the data is, what the jurisdiction is. The Internet is very international and this creates jurisdictional problems. We need to balance all of these risks.

The problem with bad cryptography is that it looks just like good cryptography. Almost all security products on the market today are insecure. Expert security testing is the only way to tell a secure product from an insecure one. If you think cryptography can solve your problem, then you don't understand your problem and you don't understand cryptography.

Poor Forensics

Forensics is a big deal. Preventing crime is a lot harder than detecting crime. In our society, we actually don't prevent crime. We detect crime after the fact, and we collect a body of evidence that can be used to convince a third party of the guilty party's guilt. We punish the guilty party and then through that punishment there is some back process towards prevention. We need more mechanisms for evidence gathering, and trusted evidence gathering, in computer systems.

Conclusion

The problem with bad cryptography is it looks exactly the same as good cryptography. I could hold up two products that use the same algorithms, the same protocols, the same buzzwords. One is secure and one isn't—and you can't tell the difference.

In our society, we tend to use the government as a regulatory body when the consumers can't make a good product decision—for drugs, for airline safety, for foods. This wouldn't actually work very well for cryptography, because the Internet is moving much too fast, and because the NSA's dual role isn't terribly trusted.

The only thing we can do is have expert testing. For example, in the absence of the FDA, if you were a smart drug user you might engage an expert lab to answer the question, "Is this a real drug or is this snake oil?"

On the other hand, we don't need to be perfect. No security system on this planet is perfect. I like to use the system of notary publics in this country "as an example". It's a completely broken security protocol, if you look at it. But it's good enough for a lot of things. "A secure computer is one you've insured." If you put a box around your risk and you've managed it, you've done a good job. We don't need perfect security; what we need is good enough security.

Finally, there are the limits of the technology. The overwhelming message of this talk is that cryptography is a mathematical tool. While it's essential for security, it doesn't automatically ensure security. It is not some magic security dust that you can sprinkle over your computer. The social problems are a lot harder than the

mathematics. It's much harder to build a secure system with people in it than it is to build a secure system with just math in it. That's the challenge we really have.

Good cryptography is like putting an enormous stake in the ground and hoping that the enemy runs right into it. You're not going to attack a system by running right into the strongest part. You can attack it by going around that strength, by going after the weak part—i.e., the people, the failure modes, the automation, the engineering, the software, the networks, etc.

I'd like you to go out into the world a little more skeptical and ask a lot of hard questions.

Bruce Schneier is president of Counterpane Systems. He is the author of Applied Cryptography *(John Wiley & Sons, 1994 & 1996), the seminal work in its field. Now in its second edition, Applied Cryptography has sold over 90,000 copies worldwide and has been translated into four languages. His papers have appeared at international conferences, and he has written dozens of articles on cryptography for major magazines. He is a contributing editor to* Dr. Dobbs Journal *where he edited the "Algorithms Alley" column, and has been a contributing editor to* Computer and Communications Security Reviews. *He designed the popular Blowfish encryption algorithm, still unbroken after years of cryptanalysis, as wall as Twofish, currently a candidate for the government's Advanced Encryption Standard.*

Schneier served on the Board of Directors of the International Association for Cryptologic Research, is a member of the Advisory Board for the Electronic Privacy Information Center, and is on the Board of Directors of the Voter's Telcom Watch. Schneier has an M.S. in Computer Science from American University and a B.S. in Physics from the University of Rochester. He is a frequent writer and lecturer on cryptography, computer security, and privacy. See http://www.counterpane.com/schneier.html for more information.