CS 161 Computer Security
Spring 2010 Paxson/Wagner Notes 4/23

Copy Protection

The purpose of this lecture is to take you on a tour through copy protection schemes, over the years. The purpose of copy protection is to enable the creator of digital content (e.g., software, music, videos) to distribute the content to paying customers, so the recipient can make use of the content, while preventing the recipient from sharing copies with others who haven't paid for it. Let's see what lessons we can learn from history in this area.

1 Music

CDs are a popular way to distribute music. The music is burned on the CD in cleartext, using a well-documented format. About 10-15 years ago, disk capacity grew enough that it became feasible and cheap enough to rip all of your CDs into digital form and store your entire music collection on your hard disk. In the past decade, network bandwidth has grown enough that it has become feasible to share your entire music collection with others on the Internet. These two facts have combined to enable widespread copying and sharing of music across the Internet: essentially any mass-market popular music you can imagine (and an awful lot of unpopular music) can be found somewhere on the Internet, for download via BitTorrent or some other means.

Of course, this situation poses a threat to the revenue stream of the music industry: if a large fraction of the population stops buying CDs, instead downloading all of their music illegally, then the industry's revenue stream will take a hit, and the industry will have less money to find and market new artists. The industry has tried several schemes to prevent people from ripping CDs into digital form and sharing them over the Internet, but none of the technical copy protection schemes have had much success. Let's look at two early schemes, which tried to ensure that CDs could be played on audio CD players (stereos, car CD players, boomboxes, etc.) but that computer CD drives could not be used to rip the CDs.

One scheme: active protection. One early attempt was to embed an autorun file on the CD, so that when you insert the CD into a Windows machine, the Windows autorun functionality would automatically execute some software found on the CD. That software would load itself into memory, detecting and preventing any attempt to access the CD drive to rip music.

However, this scheme could be defeated in a number of ways:

- Autorun is Windows-specific, so you can still rip the CD from any non-Windows machine (e.g., a Mac, a Linux machine) and then share the ripped contents.
- It is possible to configure Windows to disable autorun by default. Even easier, it turns out that if you hold down the SHIFT key while loading a CD, autorun will be disabled for that CD. As a result, this

copy protection scheme could be defeated by something as simple as holding down the SHIFT key when inserting the CD and then ripping the CD using your favorite CD-ripping software.

Even if the goal of copy protection schemes is to make copying incrementally harder (instead of providing perfect protection), this is still an awfully easy scheme to bypass.

Another scheme: passive protection. Another set of schemes attempted to exploit differences in how audio players and CD drives process multi-session CDs. To understand these schemes, you need a little background on the format of modern CDs. A CD normally contains a table of contents, a data structure that lists where each track starts and its length, and audio data for each track. In multi-session CDs, the CD contain multiple sessions; each session has a set of tracks and a table of contents. Perhaps confusingly, the table of contents for the *i*th session is cumulative: it contains information about all of the tracks in the first *i* sessions. This may make more sense, when you realize why the multi-session feature exists: multi-session CDs are normally used to let you burn a few tracks at a time. A computer CD drives typically reads the table of contents from last session and uses that to find all of the tracks on the disk.

Someone clever discovered some differences in how audio CD players vs. computer CD drives read multi-session CDs. Apparently, most audio CD players are not multi-session aware and thus read only the table of contents in the first session, while CD drives read the table of contents from the last session. In addition, audio CD players use only a few fields of the data structure in the table of contents; in contrast, many software rippers read more of the table of contents. Someone discovered that if you introduce invalid data into certain fields of the table of contents, then you can cause the firmware of many computer CD drives and ripping software products to treat the CD as invalid: attempting to read the CD would fail, e.g., with an assertion violation, or the invalid table of contents would trigger some bug in the ripper software or the drive firmware.

With this background, maybe you can see how this can be used for copy protection. The CD can be burned as a multi-session disk, where the first session has a valid table of contents and a second session has an invalid table of contents that will confuse computer CD drives. In addition, the first session's table of contents can contain invalid entries in fields that are not read by normal audio CD players.

However, it turns out this scheme can be defeated by a simple low-tech attack: if you use a felt-tip marker to carefully ink a ring along the outside of the CD, you can cover up the table of contents in the last session. This prevents the computer CD drive from seeing the invalid table of contents in the second session; it reads only the first session's table of contents, which is (by design) valid. In other words, this copy protection scheme can be defeated merely with a green marker and a steady hand.

Discussion. The reason these schemes failed is because of backwards compatibility: the format for storing music on a CD is fixed, and there is a tremendous deployed base of legacy CD players. Any copy protection scheme has to ensure that the CD can be played with legacy players, yet somehow has to prevent copying by computers that can read every bit of the contents of the CD. This is a seemingly insurmountable burden.

Today, the music industry has basically given up on copy protection for CDs and given up on trying to prevent people from ripping their CDs. To the extent that it tries to deter widespread copying, it focuses mainly on deterring sharing of copyrighted music, rather than preventing copying in the first place.

2 Video

I discussed how DVD and Blueray's copy protection schemes work.

3 Watermarking

The concept. I introduced the notion of watermarking.

A simple example. Here is a simple watermarking scheme. Let x be an image we want to watermark. Suppose the image contains n pixels; then we will represent it as a vector with n components, $x = (x_1, x_2, \ldots, x_n)$. Let's assume this is a 8-bit grayscale image, so each $x_i \in \{0, 1, \ldots, 255\}$. We'll embed a watermark, to get a new image $y = (y_1, \ldots, y_n)$, $y_i \in \{0, 1, \ldots, 255\}$. The watermark process will be under the control of a key $k = (k_1, \ldots, k_n)$, where each $k_i \in \{-1, 1\}$ is randomly chosen. To embed the watermark, we add componentwise: $y = x + k$. In other words, $y_i = x_i + k_i$.

To see how this watermark can be detected, let's first look at the dot-product of the original unwatermarked image x with the key k:

$$x \cdot k = x_1 k_1 + x_2 k_2 + \cdots + x_n k_n.$$

This is a sum with n terms, where as a rough approximation, we can say that each term is uniformly distributed on $\{-255, \ldots, 254, 255\}$ and the terms are independent of each other. Viewing each term as a random variable, each term has expected value 0 and standard deviation around 170. Thus the expected value of the sum of the n terms is 0, and the standard deviation of the sum is around $170\sqrt{n}$, since the standard deviation of a sum of i.i.d. r.v.'s grows with the square root of the number of terms in the sum. In other words, we can expect the dot-product $x \cdot k$ to be in the range $[-340\sqrt{n}, +340\sqrt{n}]$ around 95% of the time.

4 Software

Software vendors have tried various copy protection schemes, for over 30 years. The software vendor writes software and makes it available to paying customers. What prevents a paying customer from sharing it with all their friends, saving their friends from having to pay for their own copy? Normally, nothing. This is bad for software vendors, because it means they fail to capture the revenue from people who use copies without paying for them. At its worst, this threatens the business model of the software vendor and can make it uneconomic to build innovative new software.

Many schemes have been tried to try to address this problem. Let's look at a few. One challenge is that mass-market software is often distributed via CD (or some other similar medium), where for economic reasons, every copy of the CD must be identical—they aren't personalized to each individual recipient.

Host fingerprinting. If we can't embed any customer-specific information in the install CD, one alternative is for the installer to gather as much information about the specific host as it can find, and bake that into the installed software. The idea is to ensure that the software can only be used on the same machine that it was installed on. For instance, the installer could gather the Ethernet MAC address (which tends to be unique), details about the host configuration (e.g., video card), and so on. When the installed software is run, it checks that these remain the same; if any have changed, the software refuses to run.

This has many shortcomings. It doesn't prevent copying of the install CD. Someone who runs the software in a debugger can find where it is gathering information about the local host, and either stub out that code (so that the check always returns OK), or modify the code to always return the expected results. Also, it may sometimes deny legitimate users use of their software: if they install a new Ethernet card or video card, the software might stop working. Consequently, this is not a very effective approach.

Dongles. Another approach that some vendors used was to supply a small hardware dongle that the customer had to plug into their computer before running the software. In one scheme, the dongle contains a microprocessor and a signing key; the matching public key is hard-coded into the software. The software sends a random 128-bit value (the challenge) to the dongle, the dongle responds with a digital signature on this value using the private key embedded in the dongle, and the software checks the dongle's response and refuses to run if it does not receive a valid signature on its challenge. In other schemes, part of the software's functionality was implemented in the dongle. The principle here is that bits are easy to copy and share, but physical devices are less trivial to clone; if we can ensure that the software will only run when in the presence of a legitimate dongle, we can make unauthorized copying more difficult.

Dongles were unpopular among customers. If the dongle broke or stopped working, or if you lost the dongle, you couldn't use the software you paid for. Also, you had to keep the dongle plugged into your serial port, which made it harder to use your serial port for something else. In addition, in many schemes an attacker who runs the software in a debugger and finds where the validity check is being done could patch that check to always return OK without querying the dongle. This might not be possible if some critical functionality is implemented on the dongle, but that in turn has its own disadvantages: for instance, since dongles must be low-cost, their microprocessor is probably much slower than the PC's main CPU, so there is a limit to how much functionality can be offloaded onto the dongle.

Proof of physical possession. There were other attempts to tie the software to some physical item. One simple approach, used in some game software, is that the game would occasionally ask the user a question: e.g., "What object is pictured on the top of p.34 of the manual?" If the user is unable to answer correctly, the game crashes (often at a critical point in the game play). The idea is that if you download a copy of the software from some bulletin board, you won't be able to answer the question, since you won't have your own copy of the manual, so the game won't be much fun. Hopefully this will incent people to buy a legitimate copy.

With today's technology, this scheme is pretty weak. Anyone with a scanner could scan the manual and share the software along with the scanned manual, defeating the copy protection. In addition, attackers learned to run the game in a debugger, find the places where it asks the user a question, and modify the binary code to accept any answer—or to avoid asking the question at all. This leads to an unfortunate situation where the illicit copy is actually easier to use (because it doesn't bother you with questions as you are playing the game) than the legitimate copy.

License keys. I'm sure you've had the experience of installing software, and being prompted to enter a license key when you installed it. The license key is often delivered separately: e.g., on a piece of paper, by email, or printed on a sticker attached to your laptop. The software checks the validity of the license key and refuses to run without a valid license key.

What are some attacks on this scheme? One attack is to share a copy of the software and license key with others. If you are sharing them with your friend, this will probably succeed; but if you post the software binary and the license key on a public web site, to share them with the entire world, the software vendor may be able to track you down and sue you. This might deter some people from unauthorized copying.

Depending upon how the validity check is implemented, someone who reverse-engineers the validity-check code might be able to construct new license keys. One way to construct license keys is to make the first 24 bits be a random or unique 24-bit number, and the next 40 bits a MAC on the first 24 bits under some symmetric key K. However, in such a scheme, the key K must be hard-coded in the software binary (so it can check the validity of the license key), so anyone who reverse-engineers the software can learn the key

K and generate new license keys that won't be trackable to their identity. Another approach is to use digital signatures: the license key is a digital signature on a unique serial number, and the public key (but not the signing key) is hard-coded into the software. If the digital signature scheme is cryptographically secure, the software vendor can create new valid license keys but folks who reverse-engineer the software cannot. One challenge is that it is very challenging to design digital signature algorithms where the signature is short enough—no one wants to type in a 1024-bit license key—but this problem can be partly addressed by using special signature algorithms invented specifically to have very short signatures[1].

No matter how license keys are generated, an attacker can still run the software in a debugger, find the place where the license key validity check is implemented, and modify the binary to bypass this check. As a result, license key schemes are not very effective against dedicated copying.

Attack: Remove the security checks. We can see a recurring theme across all of these schemes: if the software implements some check to try to detect copying, the attacker can find the code that performs this check (e.g., using a debugger) and then patch the binary to skip that code.

This is a powerful attack, and it is hard to defend against. Software vendors have experimented with approaches to try to prevent such attacks, but all can be circumvented. Some examples:

- Perhaps the software can check whether it is running under a debugger (e.g., by periodically checking to see if any hardware breakpoints have been set, and hooking `INT 5`, which is often used by debuggers), and refuse to run if it is being debugged? But attackers can reverse-engineer the software to detect where those checks occur, patch them out, and then use a debugger at will; or attackers can use a debugger that does not leave any traces (e.g., through software emulation of the x86 instruction set).

- Perhaps the software can periodically check whether it has been modified, by checksumming itself and checking that the checksums are as expected? But attackers can find those checksum-checks and remove them from the binary, or they can modify the code and make a corresponding change to the hard-coded list of expected checksums. Even if checksums are computed on overlapping regions of the code, it still does not prevent an attacker from removing all the checks.

- Perhaps there's some more more clever way that the program can self-check its own integrity, e.g., by tying the checksum values integrally into the computation the software performs? For instance, suppose some place in the code uses the constant 27. We might rewrite the code to checksum a portion of itself (say we know that the checksum should always return 59 if the software is not being debugged), and then replace the constant `27` with the expression `calcchecksum() - 32`. As long as the program binary is not modified, this rewrite will not affect the software's behavior. If an attacker does try to modify the running binary, the checksum will be different, the software will get a different value, and this may cause the program to stop working properly. Sounds promising on first glance, but unfortunately there are more sophisticated attacks. An attacker who detects this can run the program a few times, notice that the expression `calcchecksum() - 32` always evaluates to 27 on every execution, and replace that expression with the constant 27.

- Perhaps the software should not crash immediately when it detects an attack, but silently corrupt a few data structures ever so slightly, so the program will crash much later? The hope is that this might make it harder to use a debugger to locate exactly where the checks are performed, since we cannot just wait for the program to die and then go back a few instructions. The experience in practice

[1] I've heard a rumor that Microsoft Windows license keys are constructed following such an approach, but I've never been able to confirm it.

with this approach does seem to help. It can be circumvented given sufficient patience. However, anecdotally, this does seem to make the attacker's job harder, and often slows down copying.

The bottom line is this: given sufficient patience and skill with a debugger, it is possible to defeat essentially every copy protection system that relies on embedding checks in the code. The most we can hope for is to force the attacker to work harder.

Over the past two decades, an entire community of crackers has arisen who exist solely to defeat copy protection measures and share illicit copies of software ("warez", in the jargon). The crackers compete with each other to try to crack the latest game or other software: the first ones to crack a new game get kudos and admiration from their fellow crackers, and the best crackers rise through the ranks and gain access to the most "elite" web forums and teams. This means that crackers are able to practice their cracking skills, starting out with easier-to-defeat software and gradually tackling better-protected programs as their skills and experience grows. Perversely, in some cases, the more difficult the copy protection is to defeat, the more that crackers relish the challenge. All it takes is one person to defeat the copy protection, and then they can share the cracked software with everyone over the Internet, potentially enabling thousands or millions of users to use unlicensed copies of the software. Cracked, illicit software is often shared over BitTorrent and other peer-to-peer networks and is widely available on the Internet. All of these factors make unauthorized copying very difficult to prevent.

Perspective. Does this mean we should give up all hope of building a secure copy protection scheme? Is the entire approach doomed to failure? At this point, it may be worth going back to the goals of copy protection schemes.

At its core, copy protection is about protecting the revenue of the software vendor. It does not need to be perfect. If a copy protection mechanism saves the vendor more than it costs them, then it is worthwhile. Consequently, a copy protection mechanism does not need to eliminate all copying. It does not even need to prevent large-scale copying, as long as most of the folks using the copied software would have been unlikely to pay for the software in any event. So we don't need Fort Knox level security here; it often suffice for the copy protection to be a speedbump, which makes copying just hard enough to keep unauthorized copying relatively rare.

In addition, if vendors can ensure that it takes several months before cracked copies become available, they gain a significant advantage. In some markets (like game software), most revenue is made within the first few months after initial release of the software. So, delaying the bad guys by a few months can be worth a lot of money.

The state of the art. Here is a rough overview of typical practice among the software industry today:

- **Enterprise software:** For expensive software, sold mainly to companies, many software vendors make no attempt to provide strong copy protection, and just use simple license keys. The goal is only to keep honest people honest. They figure that no company wants to break the law, and so corporate management will generally try to avoid knowingly using illicit copies. They also rely upon non-technical measures to mitigate illicit copying: for instance, the Business Software Alliance spreads the message that unauthorized copying is illegal and can cost companies a great deal, and has a whistleblower's hotline where company employees can report use of unlicensed software. In addition, services and technical support are important to many companies, which acts as an additional disincentive to use copied software, since they know that if they don't buy a license they will not be able to receive technical support or other services from the software vendor.

- **Games.** A portion of the gaming market has moved to special gaming consoles, with special hardware. The standard game platforms usually incorporate hardware-level mechanisms to make it difficult to reverse-engineer or tamper with games. Because these platforms are not a general-purpose computer, an attacker can't just fire up a debugger and observe the execution of the game software; generally, the attacker has to tamper with the hardware in some way. As a result, these platforms can offer significantly improved security against unauthorized copying.

- **Mass-market software.** Software vendors often take a pragmatic perspective: they try to make copying more difficult, but are under no pretenses that they can prevent all copying.

  Sometimes, software is written to "phone home" over the Internet and report information about the platform on which it is running. If someone tries to use the same license key to run the software on many machines, the central server can detect this and instruct the software to self-destruct. However, this can raise privacy concerns, and crackers may be able to remove the "phone home" functionality.

  Another effective approach is to design the software so that it crucially relies upon functionality that is implemented on a server in the Internet, so that the software is useless without connecting to the server. A good example would be a multi-player game, where much of the fun comes from being able to play a game against other players on the other side of the Internet.

  In some cases, vendors may apply fairly sophisticated techniques. For instance, game vendors may embed checks that have a delayed effect: a failed check might cause the game to run fine for hours, but then crash halfway through the quest. As another example, they might embed checks that are time- or date-dependent, so some checks take effect only two weeks after the software release, and some later, and so on. The idea is that crackers will find and disable some security checks, but (because crackers don't have the time or patience to do extensive play-testing, and because they often compete to be the first to crack the software so they can claim the corresponding kudos) crackers will initially miss the checks whose effect is delayed. If all goes well, shortly after the game software is released, crackers might quickly post a "cracked" copy of the software, which on first impression appears to work correctly, but actually is flaky or crashes after several hours of playing time. If the vendor is lucky, "warez" forums will be flooded with imperfect attempts at cracking the software, leading to confusion and frustration among many users who try to download illicit copies of the software and making it easier for many users to just pay for a legitimate copy.

Lessons. There are several lessons we can take away from this overview:

1. Copy protection is almost inevitably breakable; the relevant question is not whether it can be defeated, but how easy and cheap it is to defeat the copy protection.

2. The goal here is economic: just raising the bar may be enough. Preventing unsophisticated copying can still be valuable, even if we cannot prevent sophisticated attempts to defeat the protection, because it may reduce the number of users who use illicit copies of softawre.

3. Non-technical defenses can be as effective, or in some cases more effective, as purely technical defenses.