Inglorious Installers:
Security in the Application Marketplace

Jonathan Anderson, Joseph Bonneau and Frank Stajano
University of Cambridge
Computer Laboratory
15 JJ Thomson Avenue
Cambridge CB3 0FD
{jonathan.anderson,joseph.bonneau,frank.stajano}@cl.cam.ac.uk

May 22, 2010

Abstract

From mobile phones to social networks, installing and running third-party applications can be risky. Installing applications often requires running unverified, untrustworthy code with the privilege of a system administrator, allowing it to compromise the security of user data and the operating system. Once installed, applications on most platforms can access anything that a user can: a web browser can read users' e-mail and an e-mail client can access browsing history. Computer scientists have been developing systems for decades which follow the "principle of least authority," yet few consumer computing platforms adopt their techniques.

In this paper, we examine the application markets for ten computing platforms, including personal computers, mobile phones, social networks and web browsers. We identify economic causes for the wide variation in their installation and sandboxing techniques, and we propose measures to align the incentives of market actors such that providing better application security guarantees is in everyone's interest.

Contents

Chapter 1

Introduction

In this paper, we explore the markets for applications in several domains, derive economic lessons from them and provide recommendations as to how incentives can be re-arranged such that the security of users' information is in everyone's interest.

In Chapter 2, we consider the economic roles fulfilled by the various stakeholders in application markets, and their capacity to influence security properties of the system as a whole.

In Chapter 3, we present ten case studies of application markets for traditional operating systems, mobile phones, web browsers and social networks. We examine how application platforms in each of these markets provide, or fail to provide, security guarantees to both users and the system itself. We find that installing applications often requires running unverified, untrustworthy code with the privilege of a system administrator. Additionally, it can be very difficult to determine what is being installed: the installer for a CD player might also install a difficult-to-remove "rootkit" which opens the computer to infection by malware. Once installed, applications on most platforms can access anything that a user can: a web browser can read users' e-mail and a file sharing client can install a keylogger. Nonetheless, there is a glimmer of hope for users: many newer operating systems, such as those intended for mobile phones, do provide sophisticated protection of both user and system state. The important question is, why do some platforms provide these protections and not others? We believe that answers lie not in technology, but economics.

In Chapter 4, we draw out economic lessons from these case studies which explain the current state of affairs and suggest future actions. We suggest several ways of re-aligning incentives such that security for the end user is in everyone's interest. We believe that such changes can be made without damaging existing markets and business plans, with one exception: those business plans that rely on deceiving users into running software that works against the user's interest.

Chapter 2

Stakeholders

We first consider the economic roles of actors who participate in the applications market. By way of example, consider the case of a user, Alice, who wishes to run an application on her corporate smart phone. Before she runs it, it might be purchased and installed by Bob, the system administrator. This must be approved by the company's owners (or executives acting on their behalf), who also pay Alice's and Bob's salaries. The application runs on a platform developed by a major software firm such as Apple, but is distributed by a third-party vendor such as the company's wireless carrier. The corporate IT department keeps a list of approved smart phone applications, which may have been certified as "safe" by an anti-virus firm. This process may be helped or hindered by the technical details of how the application has been bundled into its installer: some techniques expose the exact operations that will occur during installation, while others allow the application to do anything that its developer might like, such as read sensitive business secrets from the phone's memory and upload them to a competitor.

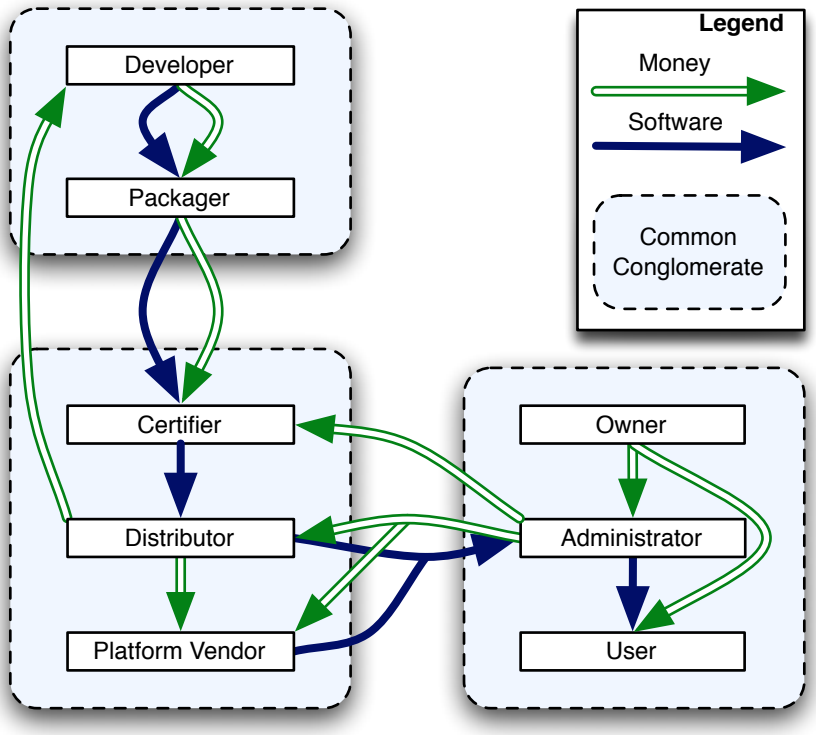


Figure 2.1: Stakeholders in the applications market.

Figure 2.1 outlines the actors in this market, including the flows of payment and software among them. Each has incentives related to application and installer security, and each can push costs on to

other actors via these flows. Multiple roles may be filled by the same entity: it is common for desktop application developers to create their own installation packages, and mobile phone vendors may operate their own App Store. In this case, their economic concerns are shared, but the external flows of money and software still exist, so the ability to transfer costs to other actors is conserved.

We consider the goals, incentives and influence over security of each of the actors in Figure 2.1, moving from the software's consumer, the User, through the Administrator and Owner to the Platform Vendor, Distributor, Certifier, Packager and finally the software's producer, the Developer.

2.1 User

In a business environment, the User, Administrator and Owner roles may be filled by discrete entities, but they are often one person in consumer settings. Nonetheless, we may consider the User as a role independently of the Administrator and Owner.

Goals

Non-administrative users have one primary goal with respect to software, and that is to be able to use it. Correct functioning may include customisation of the computing experiences via add-on applications, extensions, etc. that do not require the involvement of privileged system administrators. Software which does not function—or which prevents other software from functioning correctly—has no purpose. Inherent in this **availability** requirement is a requirement for **platform integrity**: in order for any application to function correctly, the underlying application platform must function correctly.

After availability, the User may have other security concerns; which concerns are deemed important will depend on the nature of the application. In some cases, it is very important that applications not disclose the User's data (**confidentiality**). In others, it is more important that applications not corrupt data (**integrity**). In most cases, both will be important: an accounting application should neither e-mail financial details to scammers nor modify online banking transactions to send them money.

In all cases, perfect enforcement of security policies is unlikely to succeed as a purely technical endeavour. Some enforcement will require human insight into whether or not an activity "looks right," which itself requires **transparency**: the User must be able to examine the behaviour of software.

Incentives

The User's costs are very simple: when applications are not available for the User's platform, or if installed applications misbehave, she loses time, energy and/or money.

Influence

The User's options are also very simple: unless she is also an Administrator or Owner (below), she has little power to push her costs on to any other actor. She can complain to e.g. the Owner about issues such as data loss, but such complaints may not be able to compete against other interests. Consequently, we will see security technologies in Chapter 3 which impose themselves on, rather than assist, the User.

2.2 Administrator

Goals

The Administrator is responsible for ensuring that computing platforms are available for use by the User. The Administrator's primary interest is ensuring that the system itself is protected from applications (system integrity), since fixing broken systems costs time.

Incentives

The Administrator may be employed by, or in fact *be*, the Owner, in which case protecting system integrity as well as the confidentiality and integrity of the Owner's data is of the utmost importance.

The Administrator role may also be filled by the Distributor (e.g. wireless carriers in the case of Symbian smartphones, §3.2.1) or the Platform Developer (e.g. Apple, in the case of the iPhone, §3.2.2), in which case the above assumption may fail to hold.

A rational Administrator is risk-averse, and installing software imposes costs. The installation process itself may be inexpensive, but should the software perform incorrectly or maliciously, there will be costs in investigating and repairing damage. This risk can be reduced through software verification, but that verification itself can be very expensive or, in the case of social networks with applications running on external servers, impossible.

Influence

The Administrator is likely to be responsible for purchasing software, and thus he has some financial power over the Distribution Channel, Platform Vendor and Developer. The Administrator can also push some of the cost of maintaining system integrity on to the User. This can be accomplished through policies which limit what software may be installed to a set of applications which are considered trustworthy, or at least trusted,[1] restricting what installed software is capable of doing or requiring users to personally assume risk for potentially unsafe actions (e.g. "Cancel or Allow?").

2.3 Owner

Goals

The Owner of a computing platform, like the User of that platform, is interested in provision of what the User needs to do her work, but also in the condition of her property (system integrity). She has an interest in confidentiality and integrity of the data which resides on her systems, and also the perception of her that others have (e.g. whether or not she is perceived to be trustworthy).

Incentives

The Owner's costs are similar to the User's, with one important difference: the Owner may balance differently the costs of security policies with the risks to her systems and data. These risks may be direct or indirect—an Owner whose data is disclosed in an unauthorised manner may be subject to **reputation costs**. This may mean that the Owner is willing to allow the Administrator to impose costly policies on the User that the User herself would not be pleased to accept.

Influence

The Owner holds the Administrator to account for the availability of a computing system and the confidentiality and integrity of business information.

2.4 Platform Vendor

The Platform Vendor controls the operating system and/or hardware that applications run on. This endows him with considerable influence but also imposes considerable pressure on him.

Goals

Initially, the primary goal of the Platform Vendor is growth. A computing platform is a two-sided market: its value is not simply in the features that it provides, but the applications that it supports, which is itself related to popularity in that the more widely-used it is, the more attractive it is for software developers to target it. It is because of this feedback loop of network effects that the value of a software platform grows superlinearly with the number of applications and users and the Platform Vendor initially cares more about growth, which starts the cycle, than anything else.

[1] Note that making software trustworthy *reduces* risk, whereas labelling software as trusted simply *acknowledges* risk.

Once a platform has established a stable market share, continued success depends on its ability to stave off competitors, either within the market or—in the case of near-monopoly—*for* the market [12].

Incentives

The initial growth and continued success of a software platform depends on many factors, including:

Development Costs The popularity of a platform largely depends on a cycle of adoption, which will be difficult to induce if it is difficult to program for. The Platform Vendor thus has an incentive to keep development costs as low as possible, especially before the platform has gained significant market share.

Reputation If a platform acquires a reputation for being unsafe, its growth will be stunted. For example, individuals do not use Facebook, not because of specific failings in the platform, but because of a general uneasiness with the safety of their private information. Similarly, the prevalence of unreliable device drivers in the 1990s tainted the reputation of Microsoft Windows, leading to a significant effort to "clean up" drivers (see §3.1.1).

Switching Costs Once a stable market position has been attained, loss of current users is unlikely if the cost of switching to another platform is high. Such costs can be tangible (e.g. the cost of replacing hardware and applications) or intangible (e.g. the cost of using a platform different from that of the majority of other users).

Entry Costs Market share may also be maintained where the cost to entering the market—building a new platform—is high. Facebook (§3.4.1) initially succeeded because the technology behind it was simple and inexpensive, allowing it to capture large segments of the market quickly. Today, a new firm would need to make significant infrastructure investments in order to become a viable competitor.

Price All other factors being equal, platforms must compete on price. This is especially relevant in e.g. the web browser case (§3.3), where the price of the most popular competitors is the marginal cost of production: nothing.

Influence

The Platform Vendor has significant power to shift costs onto other actors in the market, constrained by the above factors.

The Platform Vendor sets the conditions under which applications are installed and run, so he can require applications to be packaged and executed in a transparent, auditable way. Such a transparency requirement would simplify the protection of systems and makes users, which might provide a competitive advantage in terms of reputation. Such a requirement, however, comes with increased development costs both to the Platform Vendor and the Developer: rather than allowing "kitchen sink" code in an installer bundle, the system's installer must be upgraded to provide richer APIs to applications, and applications must be upgraded to use them. Nonetheless, some platforms such as mobile phone OSes (§3.2) have adopted such measures to varying degrees.

Choosing not to require such transparency may require the User, Administrator and/or Owner to accept the risk of running potentially malicious applications, or it may increase verification costs for the Certifier, potentially infinitely so.

2.5 Distributor

At the very centre of the applications marketplace is the Distributor. In some cases, the distribution channel is controlled by the Platform Vendor (e.g. the Facebook Application Directory, the Android Market), but in others, applications can be obtained from a number of channels (e.g. boxed PC software, Orkut).

Goals

The Distributor's goal is to move product: she makes her money by selling software. All of her incentives relate to how much software she can sell.

Incentives

In the case of traditional shops, software faults have no economic impact: one does not blame Best Buy for a bug in Adobe Photoshop. Online distributors may wish to add value to the distribution process through reviews or reputation systems, but the "common carrier" principle still largely applies. In more controlled channels, however, the brand of the channel and platform is tied up with the quality of software on offer; potentially harmful software released in such a channel does harm to the distributor's reputation. In this case, it is in the distributor's long-term interest to test software's security properties, so some channels—e.g. Apple, Mozilla and some social networks—actually do.

Influence

The Distributor decides which applications to sell through his distribution channel, so if he has a significant share of Users, he will have considerable influence over the Developer/Packager. Distributors like Apple and Facebook have the exclusive ability to distribute applications for their respective platforms, making Developers entirely beholden to their application security policies.

2.6 Certifier

The Certifier is responsible for ensuring that application software is safe for the Administrator to install and the User to use. This largely occurs via inspection of compiled applications, their source code or—in certain high-assurance contexts—the development process used to write the code.

In some contexts (e.g. the iPhone App Store), the Certifier, Distributor and Platform Vendor roles are filled by one entity. In others (e.g. Common Criteria evaluation of security-sensitive products for governments and financial institutions), the Certifier is an independent entity contracted by the Developer/Packager (see §2.7).

Goals

The purpose of the Certifier is to reduce the risk of installing software by analysing it and verifying certain properties, e.g. "this card game does not read the e-mail saved on the hard drive." The Certifier's goal is to be able to perform this analysis at a cost which is deemed acceptable by his potential customers, be they Developers, Packagers or Distributors.

Incentives

The Certifier is motivated by three incentives: the direct remuneration provided by the contracting party, usually the Developer or Distributor, the cost of analysis and the indirect effects of identifying (or failing to identify) flaws.

The effect of financial remuneration depends on who supplies it. Drimer et al. found that the competitive market for Common Criteria evaluations of payment systems, in which PIN entry device manufacturers pay an evaluation lab of their choice to certify their products, has driven evaluation costs down but encouraged evaluators not to look too hard for security problems that they are not explicitly contracted to look for, which could jeopardise future business [18]. When the Certifier is contracted by the Distributor—as in the iPhone App Store, where Apple employees examine submitted applications—the result is a process which protects the interests of the Distributor and its commercial partners but not necessarily those of the User [38].

The cost of analysis depends on the nature of the software being analyzed and the platform that it will run on. Standard application binaries are largely opaque, requiring enormous effort to analyze. Very often, such analysis can be shown to be theoretically impossible: one cannot, in general, determine what a piece of software will do under all circumstances, or even if it will ever finish its work [15, 27].

The Developer/Packager can employ techniques to increase transparency, however, including procedural work such as revealing the source code and how it was developed or technical work such as generating mathematical proofs that the software fulfills certain properties [32]. The Platform Vendor can also provide mechanisms for software confinement ("sandboxing" [33, 25, 11, 8, 31]) which provide limits that the software must work within.

The first indirect incentive for the Certifier is clear: should he acquire a reputation of incompetence, confidence in his work will be shattered. For instance, evaluation labs can theoretically lose the right to issue Common Criteria certifications. The Certifier's regulating body, however may be loathe to de-approve him lest confidence be lost in "the system"[18], confidence which is the *raison d'être* of certification.

A second indirect incentive for the Certifier could be—although it rarely is today—the assumption of risk, should their certification prove faulty. The feasibility of such risk management is discussed in §4.3.

Influence

An impartial Certifier, sure of his tenure—such as the NSA or GCHQ in the old Orange Book evaluation scheme—wields great influence over the Developer/Packager: unless products are produced *just so*, they will not be certified, and security-conscious Owners might choose products which have been certified. Producing products passes the Certifier's costs on to the Developer/Packager, and these costs can be considerable. Passing them on, however, can actually reduce them, as it is easier for the author of software to demonstrate its trustworthiness than it would be for an outsider. In the case of binary applications, such a cost shift can change the certification from theoretically impossible to merely expensive, reducing the uncertainty, and therefore the risk, associated with the certification.

A certifier in such a strong position may also influence the Platform Developer, passing on both costs, e.g. that of developing effective confinement and access control technology, and rewards, e.g. access to markets with a low tolerance for risk and a willingness to pay for certified software.

A Certifier which must compete for lowest-bid business, however, will merely act as a check on security practices which have been positively identified and are universally acknowledged as insupportable. Such a certifier will have little influence on the industry as a whole, but will serve to influence the Developer/Packager which fails to meet the lowest common denominator of industry "best practice."

2.7 Packager

In the open source world (e.g. Ubuntu Linux, §3.1.3), the role of Packager is often occupied by an entity distinct from the Developer. Even in closed-source software, the role is partly occupied by companies such as Flexera, which creates the popular InstallShield application.[2]

Goals

The goal of the Packager is to package software with minimum effort and maximum flexibility. If significant effort is required to learn the packaging architecture of a platform, she would at least like the process to be reproducible, so that an initial, fixed cost can be amortised over many software packages.

The Packager would like packages to be flexible enough to present a customised user experience (logos, skins, etc.) and adapt to special installation requirements, e.g. setting up licences or installing plugins.

Incentives

The cost of packaging software is non-trivial even in low-assurance configurations, such as the existing Ubuntu practice. Creating packages to work with least-privilege installers creates new costs, although in the closed-source world, the Packager can simply increase her prices.

The highest costs of such a transition would be imposed on the Packager who attempts to bundle undesired, secondary software. A transparent installer would allow the Administrator to discern the presence of e.g. a rootkit and reject the installation. This secondary software would then need to be offered to the market via the Distributor, and be subject to the normal operation of supply and demand.

[2] `http://www.flexerasoftware.com/products/installshield.htm`

Influence

The Packager is in a position to influence both the Developer and the Distributor. The former, a role which the Packager may herself fulfil, will be sensitive to the cost of packaging an application. This cost transfer is very direct. The cost imposed on the Distributor is less so, but very real nonetheless: if there is competition between Distributors, the Packager may choose not to package her software in the manner required by the Distributor.

2.8 Developer

The Developer writes the software that is packaged by the Packager, distributed by the Distributor, installed by the Administrator and used by the User. The Developer and Packager roles are often, but not always, fulfilled by a single entity.

Goals

The goal of the Developer is to write software which will be used widely, or in the commercial case, generate revenue. For some Developers, this revenue will come from sources that the User may not expect, as in social networking applications which leak such personally identifiable information as users' e-mail addresses to paying advertisers [26].

Incentives

The Developer may pay an external Packager to package his software, or he may fill that role himself. In either case, an increased cost of packaging leads directly to increased costs for the Developer.

The Developer may wish for several of his applications to share resources and application code, e.g. common software libraries. If the Platform Vendor's installation architecture does not allow this, the Developer's costs may be increased, or he may not be able to write software that does what he wants.

Influence

In the end, the Developer's costs are exposed to the Distributor, Owner and/or Administrator as the price of the software. In the case of freely-available software, this increased cost may come in the form of less functional or less available software — it may cease to be worth the Developer's time.

Chapter 3

Case Studies

The protection afforded to users and the system itself varies from operating system to operating system. We present a series of case studies, in which we examine the security properties of traditional operating systems, mobile phone platforms, Web browsers and social networks. These properties include:

Authority What authority do applications have? Can they modify any user data, or are they confined to "sandboxes?" Can they—or their installers—modify the Trusted Code Base (TCB) of the system?

Transparency What properties of application software can actors in the marketplace observe or measure? Can actors verify that the application respects some security policy? Can the Administrator audit the activity of application installers? Can the user observe which applications are currently running, and what data they are allowed to access or modify?

Warranty Does any actor verify security properties of the application on behalf of another?

3.1 Desktop Operating Systems

The traditional operating system is a multi-user operating system that runs on general-purpose hardware, usually x86-, IA64- or amd64-based. Microsoft Windows, Apple's Mac OS X and Canonical's Ubuntu Linux distribution are all examples, incorporating traditional security features such as process isolation, *Discretionary Access Control* (DAC) and optionally, *Mandatory Access Control* (MAC).

Installing and running applications on each of these operating systems is different, but there are important common features. When installing software, it can be very difficult to determine what is being installed where, as all three provide installers which can run arbitrary code with system privilege. Once the installation is complete, applications run with the full *ambient authority* of the user, and they can register themselves to start automatically upon login without user knowledge, much less permission.

3.1.1 Microsoft Windows

As the clear market leader in consumer operating systems, Microsoft Windows has fallen victim to many software defects. Some of these defects originate from accident, both others from malice. Over the past decade, Microsoft has made significant strides in protecting the Windows operating system from defective drivers and applications; unfortunately for users, little has been done to protect users from malicious applications.

Applications

The first version of Microsoft Windows to ship with a Microsoft-supplied installer was Windows 2000.[1] Before Windows Installer, a software installer was a program like any other. These proprietary installers

[1] "Windows Installer 1.0 SDK (x86)", Microsoft Download Center, `http://www.microsoft.com/downloads/details.aspx?FamilyID=105dfc41-801e-4441-aa75-7696d794f109`

were not concerned with the integrity of anything other than what they were installing, so they could — and often did — overwrite core system components [30].

Since Windows 2000, Microsoft has steadily improved its ability to protect the operating system's core components. In software installation, however, little has changed since 1999, and users still receive no protection from malicious or disingenuous installer packages even today.

System Protection Over the past decade, Microsoft has improved their system protections steadily and effectively, starting with Windows File Protection (WFP) [30].

WFP is a system service which checks core system files for unauthorised changes — those that do not come from Microsoft — and replaces corrupted files with "known good" versions. Before WFP, applications would often replace system DLLs with customised versions, a behaviour that led to "DLL hell" when the new files broke existing functionality in other applications or the system itself.

With Windows Vista, Microsoft introduced another system for protecting the operating system: the Windows Integrity Mechanism.[2] Windows Integrity is based on the Biba model [7]: system objects and subjects (e.g. processes) are labelled according to integrity levels (Untrusted, Low, Medium, High and System), and policies can be applied such as "no write up", in which less-trusted subjects cannot write to more-trusted objects. System administrators, assigned a High integrity level, cannot overwrite system components which have been assigned a System integrity level.

This new integrity model is not brought to bear on the problem of application installation, however. Windows Installer runs at two integrity levels: the UI launched by an authenticated user runs at Medium, and the system backend runs at the System integrity level.

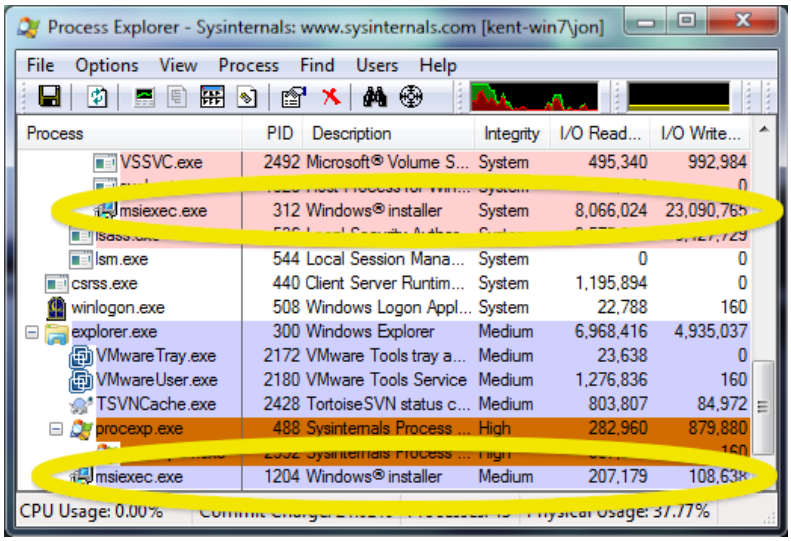


Figure 3.1: Windows Installer running at both Medium and System integrity levels.

Figure 3.1 shows that, of these two components, it is actually the System-level one that writes the application to the filesystem (the System version of `msiexec.exe` writes ≈20 MB to the filesystem, whereas the Medium version writes just ≈100 kB). Thus, the Windows Integrity protections do not apply to software installation.

User Protection Since the introduction of Windows Installer, other commercial installers are encouraged to build on top of, rather than re-implement its functionality.[3] WI is, thus, the standard means of installing software. WI uses a database-like system to enumerate items to be installed, e.g. the application's directory structure,[4] which could allow application installations to be easily audited, but it also

[2] "Windows Integrity Mechanism Design", MSDN Windows Development Center, `http://msdn.microsoft.com/bb625964.aspx`

[3] "Windows Installer," MSDN Windows Development Center, `http://msdn.microsoft.com/cc185688.aspx`

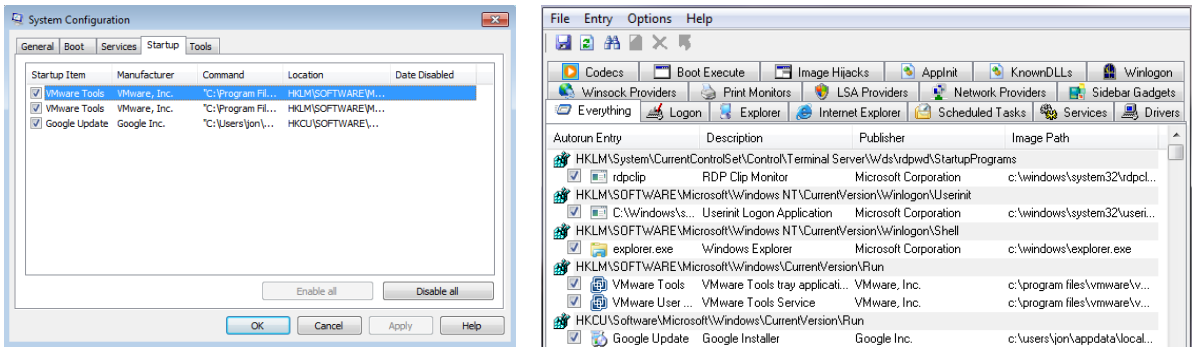
[4] "Directory table", Windows Installer Guide, `http://msdn.microsoft.com/aa368295.aspx`

allows custom code to be run, sometimes with system privilege.[5]

Application installers are not transparent: application installers do not carry a complete list of files that might be installed, Windows Installer provides no facility for users to see exactly what is happening to their computer, and users can easily be tricked into installing unwanted software. One of the most famous examples is the popular peer-to-peer filesharing client Kazaa, which installs spyware along with the application that the user expects [36].

Without installer transparency, users must trust to reputation, hoping that "legitimate" software will not trick them in this way. Unfortunately, this trust is sometimes ill-placed. Some examples are relatively benign: iTunes also installs QuickTime by default, and installing ClamWin (open-source antivirus software) also installs the Ask.com toolbar.[6] In these cases, users can "opt out" of the additional software, but not all software providers are so magnanimous: no less a well-known company than Sony installed a rootkit on Windows PCs to prevent users from copying CDs, causing stability problems and opening PCs to attack by other malware.[7]

During installation, applications may register themselves to be run automatically on startup without the user's knowledge or permission. In the best-case scenario, these applications exact a cost on the PC's performance; in the worst, they could be sending spam, running phishing websites or scanning the hard drive for passwords and banking details. Figures 3.2(a) and 3.2(b) show Windows tools that can be used to control these programs by disabling their Registry entries; the existence of these tools points to a need for more systematic control over automatic execution.



(a) `msconfig` (b) `autoruns`

Figure 3.2: Tools for configuring Windows startup applications.

Device Drivers

One area in which Microsoft has made significant effort and progress is the reliability of device drivers.

When Microsoft studied Windows NT reliability in the late 1990s, they found that 44% of NT crashes were caused by faulty drivers [30]. When an application crashes in a modern operating system, the source of the fault (the offending application) is quickly identified. Device driver defects, on the other hand, are revealed through OS kernel panics — in Windows, the famous "Blue Screen of Death."

In order to improve the reliability and reputation of Windows, Microsoft created the Windows Hardware Quality Labs (WHQL). This laboratory certified hardware and drivers as meeting a minimum standard of quality known as "Windows Logo" testing. This testing included running the Driver Verifier tool, which stressed hardware drivers by running them in low memory conditions with random faults [30]. Certified drivers could be expected to cause OS crashes less often than uncertified ones, so installing an uncertified driver required clicking through a dialog such as the one in Figure 3.3. Additionally, Windows 2000 could be configured by the System Administrator to reject all unsigned drivers, although this option is not enabled by default. Even so, the Windows Logo program makes it more difficult to hide malware

[5] "Custom Action Security", Windows Installer Guide, `http://msdn.microsoft.com/aa368073.aspx`

[6] `http://www.clamwin.com/ask/`

[7] "Real Story of the Rogue Rootkit", Bruce Schneier, Security Matters, `http://www.wired.com/politics/security/commentary/securitymatters/2005/11/69601`

in device drivers: there are much easier ways to get malware onto a Windows PC than going through the Windows Logo process.

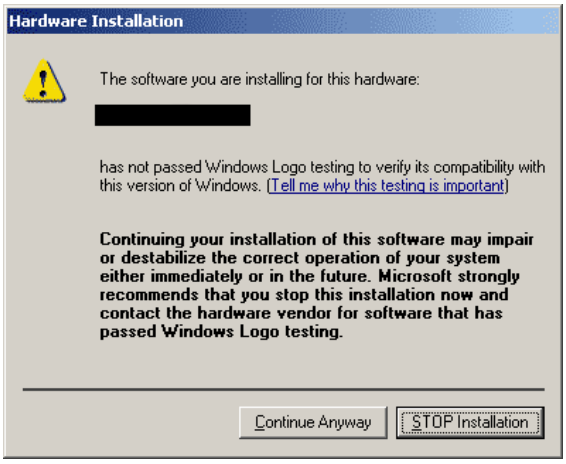
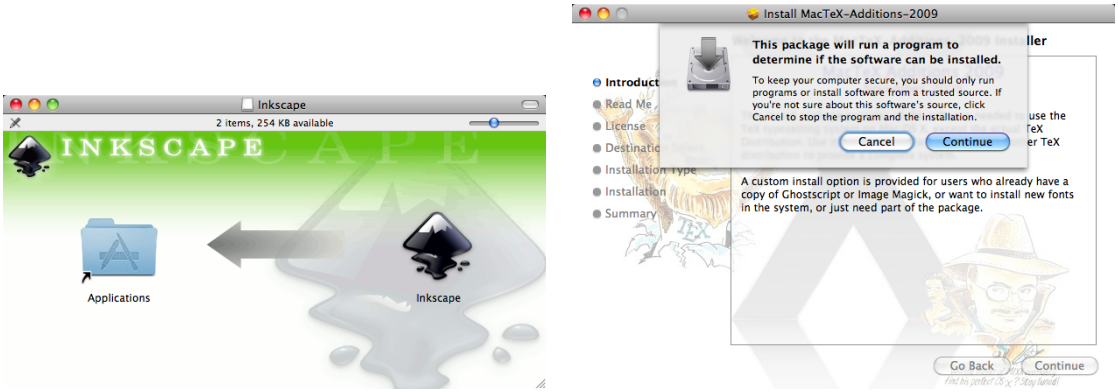


Figure 3.3: Attempting to install an uncertified device driver.

In 2006, Cook et al. published work on proving termination in certain contexts [14]. This work found non-termination errors in 5 of the 23 standard Windows device drivers included with the Device Driver Development Kit. The software used to find these errors has been integrated into the Static Driver Verifier tool, so all newly certified or otherwise tested drivers benefit from it.

3.1.2 Mac OS X

NeXTstep, the precursor of today's Mac OS X, specified that applications should be installed in *bundles*, directories which contain both a binary and the resources that it requires.[8] This pattern was continued in Mac OS X, and many applications today are still installed by the *manual install* method:[9] dragging them from a downloaded file to the system Applications directory, as shown in Figure 3.4(a).



(a) Manual install

(b) Managed install

Installing software in this way requires a user to be a member of the UNIX `admin` group only if the application is to be installed in `/Applications`: application bundles can be placed anywhere in the filesystem, including user home directories. Even if the applications are installed to the system applications directory, no installation code is executed: this operation is simply moving a directory. In

[8] "NXBundle", NeXTstep 3.3 Developer Documentation Manual, `http://www.cilinder.be/docs/next/NeXTStep/3.3/nd/GeneralRef/03_Common/Classes/NXBundle.html#index.html`

[9] "Software Delivery Guide", Mac OS X Reference Library, `http://developer.apple.com/mac/library/documentation/DeveloperTools/Conceptual/SoftwareDistribution/`

addition to providing good security properties, it has also been commended as good security usability, since it requires a very clear expression of user intent [45].

As early as Mac OS X 10.1, however, applications were including custom installers which could require acceptance of a license or allow users to select which parts of a large application suite they wished to install.[10] Apple, like Microsoft before them, then unified package installation by creating a standard system installer, which shipped with Mac OS X v10.2. This installer, appropriately named Installer, supported the *managed install* mode of *component packages* and *metapackages*. OS X v10.4 later added *distribution packages*, which change the semantics of metapackages slightly[9], but not in security-relevant ways. In addition to system requirements (e.g. "must be an Intel Mac") and volume requirements (e.g. "must have 4 GB free"), Installer allows packages to specify arbitrary binaries which run before and after an installation and/or upgrade. Figure 3.4(b) shows Installer in action, about to run a *preflight* (pre-installation / pre-upgrade) binary.

Mac OS X 10.5 and later include a feature called Sandbox, which uses the TrustedBSD MAC framework to restrict application behaviour.[11] Installer uses this feature of Mac OS X to confine portions of the application installation process. We demonstrated this using the DTrace utility, which allows running Mac OS X kernels to be instrumented. We instrumented the kernel while running the `sandbox-exec` program and discovered that the kernel function used to put a process into a sandbox is `mac_cred_label_update()`. We wrote a simple DTrace script, shown in Figure 3.4, which prints kernel and user stack traces whenever this function is called inside the kernel.

```
::mac_cred_label_update:entry
{
    trace(execname);
    stack();
    ustack();
}
```

Figure 3.4: DTrace script.

We then ran the script while running the `sandbox-exec` program, producing the output in Figure 3.5, and `installer`, producing the output in Figure 3.6. In both cases, we found that `mac_cred_label_update()` was used to make changes to the process MAC labels, and given the identical kernel stack traces in both cases, we infer that `installer` is using the Sandbox feature to constrain parts of its execution.

```
CPU     ID                    FUNCTION:NAME
  0  11972      mac_cred_label_update:entry   sandbox-exec
              mach_kernel‘kauth_cred_label_update+0x4b
              mach_kernel‘kauth_proc_label_update+0x2c
              0x1867758
              0x18655a5
              mach_kernel‘__mac_syscall+0xf0
              mach_kernel‘unix_syscall64+0x269
              mach_kernel‘lo64_unix_scall+0x4d

              libSystem.B.dylib‘__sandbox_ms+0xa
              foo‘0x100000be2
              foo‘0x100000a10
              foo‘0x4
```

Figure 3.5: DTrace output when `sandbox-exec` runs.

Once installed, applications execute with the user's ambient authority unless they take care to confine themselves. Google Chrome is one well-known application which uses the Seatbelt feature to confine

[10] "How to install Office v. X for Mac OS X (10.1)", Microsoft Help and Support, `http://support.microsoft.com/kb/311939`

[11] "Security Overview", Mac OS X Reference Library, `http://developer.apple.com/mac/library/DOCUMENTATION/Security/Conceptual/Security_Overview/Concepts/Concepts.html`

```
CPU     ID                    FUNCTION:NAME
  1  11972      mac_cred_label_update:entry   mdworker
              mach_kernel`kauth_cred_label_update+0x4b
              mach_kernel`kauth_proc_label_update+0x2c
              0x1867758
              0x18655a5
              mach_kernel`__mac_syscall+0xf0
              mach_kernel`unix_syscall64+0x269
              mach_kernel`lo64_unix_scall+0x4d

              libSystem.B.dylib`__sandbox_ms+0xa
              libSystem.B.dylib`sandbox_init+0x182
              mdworker`0x1000158e9
              mdworker`0x1000080ed
              mdworker`0x100001034
              mdworker`0x3
```

Figure 3.6: DTrace output when `installer` runs.

its sub-processes, but the top-level process still retains ambient authority. If Sandbox becomes more popular, enough applications might expect to run in a least-privilege environment that Mac OS could start enforcing Sandbox restrictions by default. Right now, however, all applications are executed with ambient authority.

Applications can also register themselves to be run on user login via the Mac OS setting "Login Items," shown in Figure 3.7 or standard UNIX mechanisms such as shell initialisation scripts (e.g. `.bash_profile`). The former is easier for users to administer than Windows equivalents, since it is part of the standard System Preferences interface, but the latter is just as esoteric as its Windows equivalents, and has the additional difficulty that scripts are full programming languages: malware can embed its startup commands via subtle changes in seemingly innocuous statements.

3.1.3 Ubuntu Linux

The popular Ubuntu Linux distribution uses the Debian package management system for installing applications, system services and even the Linux kernel.

Once installed, applications typically run with full user authority or root privilege, unless a Mandatory Access Control mechanism such as Ubuntu's (optional) SELinux support[12] is installed or the applications themselves are designed for privilege separation [2, 33, 8]. Since users are not protected against applications, let us consider the protection of the system itself.

To modify the core system as it does, the Ubuntu package installer must be able to perform incredibly powerful actions. Protection against malicious activity is ostensibly provided though technical mechanisms, but in reality, protection is derived from the vetting of packagers rather than packages.

Checking Packages

A Debian package contains a list of directories to be created and files to be installed (an example is provided in Table 3.1), and the package management system enforces the invariant that no two packages may provide the same file.[13] For instance, if a user attempted to install a game package which contained a file called `/sbin/init`, the package manager would inform them of the conflict between it and the core system package `upstart`.

This enforcement can break down in practice, however, because packages may also contain configuration scripts, which are executed by the installer before and/or after the package's files are extracted. After installation, many applications run with user authority, so they can modify a user's files, but they cannot modify the operating system itself. During installation, however, a package's configuration scripts

[12] `https://wiki.ubuntu.com/SELinux`

[13] The system is complicated by configuration files, which are treated somewhat differently; for the purposes of this discussion, however, the simple "one package per file" model is sufficient.

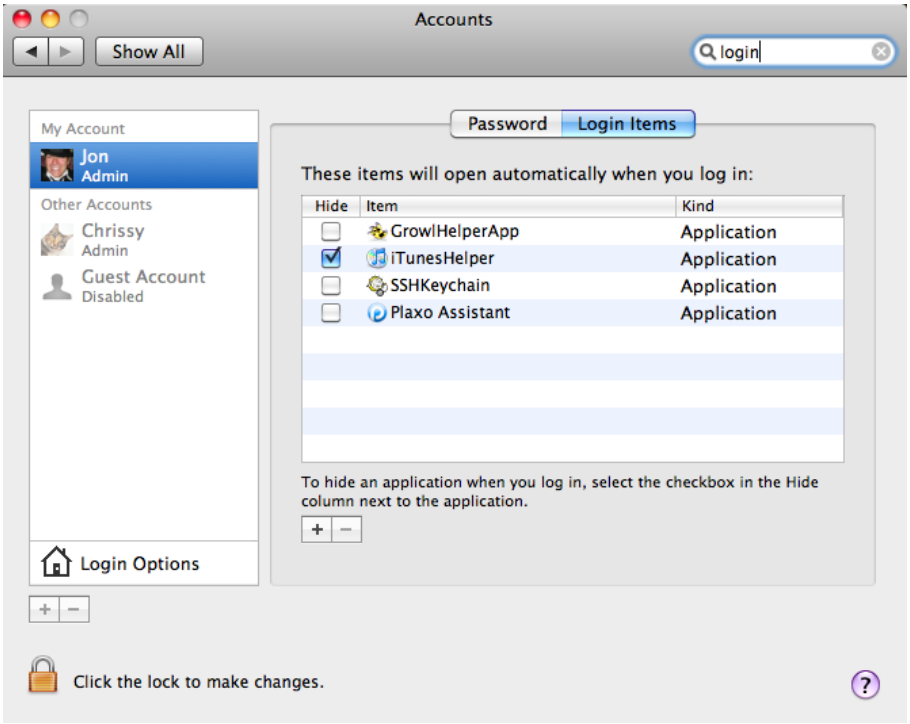


Figure 3.7: Login Items in Mac OS X.

run with system privilege, meaning that they can modify anything. In some configurations,[14] users are permitted to install packages without system privilege; configuration scripts in this scenario run with more privilege than the user would ever have access to.

To determine what these configuration scripts are doing, we downloaded the complete list of 27,499 Ubuntu Jaunty packages from `http://packages.ubuntu.com/jaunty/`, chose 200 at random using a uniform probability distribution and downloaded their Ubuntu sources. We then manually inspected all `preinstall` and `postinstall` scripts in these packages, and compiled a list of post-install actions which they take that affect the system as a whole;[15] this list is shown in Table 3.2.

Of the many system-level actions taken by post-install scripts, most can be classified as notifying the system that its state has changed in some important way: when new configuration files have been installed, system services may need to be restarted; when new dynamic libraries are installed, indexes must be updated. Some actions are unusual because they are powerful: many packages install system services which will henceforth run with system privilege, and the `bluez` post-install script can create device nodes (system interfaces to hardware devices) because it is responsible for interfacing with Bluetooth hardware.

Of the 200 packages that we surveyed, we found one abusing the post-install script facility to do something which ought to be managed by file lists and constrained by the "one package per file" policy. That example is the `pulseaudio` package, which creates a configuration link in a system directory that is owned by the `sgml-data` package. This is not a malicious act, but it is possibly an unsafe one: eschewing the explicit file creation API in favour of a less auditable mechanism could lead to a difficult-to-resolve conflict between packages. This conflict would be difficult to resolve because only one of the packages involved would explicitly state its relationship to the file, and this is exactly the sort of conflict which the Debian package manager was designed to mediate.

If this package had, in fact, come with a malicious post-install script, extraordinary damage could be done to a Ubuntu system, and because the actions taken by these scripts are not audited, it could be very difficult to pinpoint the cause of the damage. The ease with which the "one package per file" policy

[14] Using the PackageKit framework, `http://www.packagekit.org/`.

[15] Only eight packages took pre-installation actions which affected the system as a whole, all of which are represented by similar or inverse post-install actions.

/.	/usr/share/doc/bzip2
/bin	/usr/share/doc/bzip2/copyright
/bin/bzip2	/usr/share/doc/bzip2/changelog.gz
/bin/bunzip2	/usr/share/doc/bzip2/changelog.Debian.gz
/bin/bzcat	/usr/share/man
/bin/bzip2recover	/usr/share/man/man1
/bin/bzexe	/usr/share/man/man1/bzip2.1.gz
/bin/bzgrep	/usr/share/man/man1/bzexe.1.gz
/bin/bzmore	/usr/share/man/man1/bzgrep.1.gz
/bin/bzdiff	/usr/share/man/man1/bzmore.1.gz
/bin/bzegrep	/usr/share/man/man1/bunzip2.1.gz
/bin/bzfgrep	/usr/share/man/man1/bzcat.1.gz
/bin/bzless	/usr/share/man/man1/bzip2recover.1.gz
/bin/bzcmp	/usr/share/man/man1/bzcmp.1.gz
/usr	/usr/share/man/man1/bzdiff.1.gz
/usr/share	/usr/share/man/man1/bzegrep.1.gz
/usr/share/doc	/usr/share/man/man1/bzfgrep.1.gz
	/usr/share/man/man1/bzless.1.gz

Table 3.1: Files and directories installed by the `bzip2` package (Ubuntu Jaunty).

Freq.	Action	Examples
47	Update system indexes	Dynamic linker bindings, Python modules
12	Alter system-level, namespaced config	Apache `sites-available` entries
9	Alter users or groups	
9	Install system-wide, indexed content	Emacs packages, GNU info pages
8	Restart system services	Reload Apache
8	chown/chmod	
4	Alter system-level, shared config	PHP configuration file
3	Copy/link public files to private space	
2	Create device nodes	
2	Run private indexing tool	GCJ Java class index
2	Configure a shared database server	Adding a new DB to MySQL
2	Run application immediately	`netcfg`, a network configuation GUI
1	Create shared directories	System `icons` directory
1	Register “reboot required” notification	
1	Create system configuration symlinks	Overwrite system SGML configuation

Table 3.2: Actions taken by Ubuntu `postinstall` scripts.

can be worked around shows that the enforcement mechanism exists to prevent accidental changes to system state, not malicious ones.

Checking Packagers

Since technical mechanisms do not protect systems from malicious packages, users must simply trust that the packages made available by the Ubuntu project are not malicious.

Software packages often come to Ubuntu users by a circuitous route. The Ubuntu project uses Debian as its basis, which itself incorporates open-source software from many sources. One interesting property of this model is that software *packagers* are often different from software *authors* (see §2.7 and §2.8).

For instance, copyright to the compression software bzip is held by Julian R Seward.[16] The current version of the software, as released by Mr Seward, is 1.0.5. This, in Debian parlance, is known as the *upstream* version. Debian maintainers take this software and add scripts to turn it into a Debian package, `bzip2-1.0.5-1`.[17] This version of the software is then taken by the Ubuntu project and repackaged, often automatically, into the version which can be downloaded by Ubuntu users (in this case, `bzip2-1.0.5-1ubuntu1`). Configuration scripts may, thus, come from any of three sources: the original author, Debian maintainers or Ubuntu maintainers.

Further complicating this system is the fact that over 19,000 of the 27,500 available Ubuntu packages are maintained by a community of 144 (as of writing) volunteers known as the "Masters of the Universe."[18] Becoming a MOTU requires Ubuntu membership, so every MOTU has made "sustained and significant contributions" to Ubuntu over a period of at least six months.[19] The security of almost every Ubuntu system in the world, then, depends on this process weeding out malicious contributors and on loyal MOTUs auditing those 19,000 applications and their configuration scripts.

Once installed, applications run with the user's ambient authority. There are a variety of scripts and directories which can be used to execute applications on login. Exact details vary among shells and desktop environments, but in all cases scripts are used to start the execution of programs, and these scripts can be modified to hide malware startup.

3.2 Mobile Phones

As mobile phones become increasingly "smart," they require increasingly sophisticated operating systems. Today's smart phones (and "super phones" in Google parlance) run full-fledged operating systems, often closely related to traditional desktop OSs. These operating systems are able to run powerful applications downloaded from portals which are often operated by the platform vendor or mobile network operator.

As more and more personal computation occurs on smart phones rather than traditional computers, concern grows about the security of these operating systems[20] and some call for improved testing of mobile applications.[21] We seek to answer this question — and others — by considering the cases of three popular mobile phone operating systems: Symbian, originally written by Nokia but now managed by a non-profit foundation, Apple's iPhone OS (based on Mac OS X) and Android OS (based on Linux).

We find that these OSes provide significantly more protection to both users and the system than their desktop counterparts, but that there are very important differences among them in terms of a user's control over applications.

3.2.1 Symbian

Symbian OS v9.1, released in 2005, introduced *Platform Security* to protect phones and their users against malicious software [39]. Prior to this version of the operating system, an application loaded onto a Symbian phone was fully trusted: any application could make any system call, and there were no access

[16] Taken from `http://www.bzip.org/`

[17] See `http://packages.debian.org/lenny/bzip2`.

[18] `https://launchpad.net/~motu`

[19] `https://wiki.ubuntu.com/Membership`

[20] "Malicious Mobile Apps a Growing Concern," ZDNet Mobile Security & Innovative Technologies Blog, `http://community.zdnet.co.uk/blog/0,1000000567,10015074o-2000440756b,00.htm`

[21] "Why Don't Mobile Application Stores Require Security Testing?", `http://blogs.gartner.com/neil_macdonald/2010/02/03/`

control checks.[22] With the addition of the Platform Security features, applications could be confined using a system of certification and permissions.

Certification

Symbian OS 9 requires nearly all applications that perform operations such as interacting with user data to be certified (see Permissions, below). Developers submit their applications to a "Test House", which conducts a verification process. How much verification occurs depends on what permissions the application requires; applications which wish to modify system data require more verification than those which only access user data. If the verification process finds no evidence of error or malice, the application is digitally signed with a key whose chain of trust is rooted in Symbian (now the Symbian Foundation). Uncertified applications may be used for development, but they are cryptographically bound to a particular phone. Applications may only be installed widely if they have been certified.

This verification process cannot, of course, be perfect, and applications have been certified which later turned out to be malware. The certificate revocation scheme was employed to deactivate instances of this application, but many carriers disable these checks because of the network traffic they cause.[23]

Permissions

When applications are signed, the resulting package includes a list of *permissions* which the application is trusted to use. The Symbian literature refers to permissions as *capabilities*, but we will use the term permissions to avoid confusion with the traditional OS concept of object capabilities [16, 35]. There are three categories of permissions that an application can be granted: "basic," "extended" and "phone-manufacturer approved."

Basic permissions, which include permissions such as **Location** and **ReadUserData**, allow applications to read or write user information. Symbian documentation distinguishes "private" user data such as "personal and sensitive images" and "'secret' notepad memos, PIN numbers and passwords" from "public" user data such as "images or videos from 'neutral' subjects" and "ordinary scribbles," a very coarse-grained distinction. The former should be stored in `/private/<Executable ID>/`, whereas the latter may be written anywhere else in the filesystem, other than the special directories `/sys` and `/resource` (these directories may only be written to by applications with the phone-manufacturer approved permission **TCB**.

Extended permissions allow applications to read or write system state; they include the basic set of permissions as well as lower-level permissions such as **PowerMgmt** and **ReadDeviceData**. Applications which request extended permissions undergo more testing than those requesting only basic permissions.

Phone-manufacturer approved permissions included all of the extended permissions, plus very low-level permissions such as **NetworkControl** and media-related permissions such as **DRM**. In order for an application to be granted these permissions, approval must be obtained from the phone's manufacturer; the ability to modify network protocols or circumvent DRM could impact significantly on mobile carriers.

Exceptions

Manufacturers can choose to designate some permissions as user-grantable. The software installation tool can prompt the user concerning these permissions; if granted, the application will have them as long as it is installed. If an application only requires permissions which may be granted by the user, it need not be certified by a Test House. Despite the availability of user-grantable permissions, however, applications cannot define custom permissions, so permission to access one type of user data allows access to all other types as well.

Instead of this "blanket permission" model, a very select number of permissions can be granted by the user on a per-use basis; the availability of this approach is manufacturer- and possibly device-specific.

[22] "Platform Security (Fundamentals of Symbian C++)", Symbian Developer Documentation, `http://developer.symbian.org/wiki/index.php/Platform_Security_(Fundamentals_of_Symbian_C++)`

[23] `http://secblog.symbian.org/2009/07/16/signed-malware-revoked/`

3.2.2 iPhone

The iPhone was announced by Apple Computer in 2007. Its operating system, iPhone OS, is based on Mac OS X and inherits much of its security architecture, including the MAC components from the TrustedBSD project.[24] Since the iPhone has no backwards compatibility to maintain, it uses some this architecture in ways which are theoretically possible but practically unworkable in Mac OS X. This includes mandatory digital signatures from Apple, restricted installation and application sandboxing.

Digital Signatures

The iPhone OS requires that applications be signed by their authors, and also from Apple:

> On iPhone OS, [...] applications not signed by Apple simply will not run[24].

This allows Apple to have control over which applications run on the iPhone, providing Apple with the opportunity to check all applications for security vulnerabilities (as well as features that might compete with the features or business models of Apple and its partners).

Installation

Applications are installed to the iPhone in a manner that resembles the "drag and drop" method of Mac OS X's *manual install*. Each application has a directory with a name that is meaningful to the device rather than a human, e.g. `17D0A5AE-8D11-47C8-B4B0-BEF13C6F4BB2`.[25] This directory contains subdirectories such as `Documents` and `Library` which, in Mac OS X, would be found in a user's home directory. This approach ensures that application installation does not overwrite other applications or core system components.

Runtime

iPhone applications are confined using the Mac OS X Sandbox mechanism. Each application is given a unique policy file which restricts the files and other system resources which it can access. This sandbox prevents malicious applications from modifying system state, except through available APIs. Unfortunately for users, though, signed applications do have access to personal information which they may not need, such as phone numbers, e-mail addresses, Safari searches and GPS locations [38].

Vetting

Unlike the Android Market, applications must be approved by Apple before they are placed in the App Store. These has led to some controversy, as when Apple did not approve the Google Voice application, a potential competitor to Apple's wireless carrier partners,[26] or more recently, when Apple purged the App Store of sexually explicit applications (but not those from "established brands" like Playboy).[27]

During the Google Voice controversy, Apple released a statement to clarify how the App Store approval process works. Apple's 40 app reviewers check all submitted applications for "buggy software, apps that crash too much, use of unauthorised APIs [...], privacy violation, inappropriate content for children, and anything that 'degrades the core experience of the iPhone' "[28] before they are signed and added to the App Store.

Such scrutiny is very difficult, however, and applications do gain approval which ought not. One application, Google Mobile, makes use of the unauthorised APIs mentioned above,[29] which could be detected by automated machine inspection, yet it passed muster. Questions which must be answered by

[24] "Security Overview", iPhone OS Reference Library, `http://developer.apple.com/iphone/library/documentation/Security/Conceptual/Security_Overview`

[25] "Running Applications", iPhone Development Guide, `http://developer.apple.com/iphone/library/documentation/Xcode/Conceptual/iphone_development/120-Running_Applications/running_applications.html#//apple_ref/doc/uid/TP40007959-CH6-SW2`

[26] "Google Voice App Rejection: AT&T Blames, Apple Denies, Google Hides", `http://www.wired.com/epicenter/2009/08/apple-att-and-google-respond-to-feds-on-google-voice-rejection/`

[27] "Apple VP attempts to explain double standard for risqu apps", ars technica's Infinite Loop, `http://arstechnica.com/apple/news/2010/02/apple-vp-attempts-to-explain-double-standard-for-risque-apps.ars`

[28] "Apple sheds light on App Store approval process", `http://news.cnet.com/8301-13579_3-10315328-37.html`

[29] "Google admits breaking App Store rules", `http://news.cnet.com/8301-13579_3-10108348-37.html`

humans are even more difficult to manage: according to the statement referenced above, each member of the vetting team vets, on average, 85 applications per day. Assuming an eight-hour work day, that means that each member of the App Store team vets one application every six minutes[28].

3.2.3 Android

Android is an open-source operating system for mobile phones, developed by based on Linux and developed by the Open Handset Alliance. Android applications can be very powerful; third-party developers have access to the same APIs as "core" applications.[30] Nonetheless, applications are both transparent in the rights they can use and rigidly confined; applications cannot access user data, other applications or system state without permission.[31]

Installation and Permissions

The package installer is an important part of Android's security model. Unlike many traditional operating systems, Android only prompts users about applications permissions at install time. This is made possible by a static permissions model.

When an application is installed, Android reads a *manifest*—an application description—from the application bundle. This manifest may list *permissions* which the application requires in order to provide functionality. Permissions are required to "do anything that would adversely impact the user experience or any data on the device." The list of standard system permissions includes permission to pair and/or connect to Bluetooth devices, send SMS messages and set the phone's wallpaper. The package installer will prompt the user for their assent to these permissions, as shown in Figure 3.8. If the user chooses to continue the installation, the application will be granted the requested permissions until it is removed from the phone. Once installation is complete, no alterations to the application's set of permissions are possible.

Figure 3.8: Android installation prompt.

On installation, each application is assigned a unique Linux user ID. This means that Linux's standard Discretionary Access Control (DAC) mechanisms can be used to control applications' access to user data, system state and each other — files written by an application are, by default, only visible to that application. Applications can create files with the `MODE_WORLD_READABLE` and/or `MODE_WORLD_WRITABLE` flags, rendering them accessible to all other applications. Two or more applications can request to share a user ID using the `sharedUserId` manifest attribute, but they will only be granted it if they both request it, and if both applications have been signed by the same private key.

[30] "What is Android?", Android Dev Guide, `http://developer.android.com/guide/basics/what-is-android.html`

[31] "Security and Permissions", Android Dev Guide, `http://developer.android.com/guide/topics/security/security.html`

Digital Signatures

All Android applications must be digitally signed by their authors. This provides data integrity, and also allows trust relationships to be established between applications (e.g. sharing a user ID). Unlike Apple's iPhone (see §3.2.2), applications do not need to be signed by the OS vendor. Consequently, software need not be approved by a single distribution channel; although a centralised "Android Market" does exist, users can install software written by anyone and obtained from anywhere.

Runtime

When applications run, security restrictions are enforced by the Android Java runtime, the Dalvik virtual machine. Although, for performance reasons, applications can include native components compiled to run directly on the phone's CPU, requests to perform privileged operations (those requiring permissions) must be served by the Dalvik VM.

Vetting

What Android's installation and runtime checks *cannot* protect users from is phishing, in which users voluntarily give up secrets to applications which have permission to contact arbitrary servers on the Internet. Applications do not undergo any security vetting before entering the Android Market, although they can be removed from it if, for instance, they are accused of phishing by banks, as happened early this year.[32] The applications in question were removed, not because any security allegations were proven, but because the names of banks were used without permission, violating the market's Content Policy.[33]

3.3 Web Browsers

Modern web browsers have evolved into extremely complex pieces of software which share many features with traditional operating systems. At a minimum, they must manage the isolation of processes (web applications) competing for resources like memory and scheduling of JavaScript threads, enforce separation between different sites, perform access control checks on network and file operations, and maintain persistent storage.

Installing plugins or extensions (together called add-ons) is directly analogous to installing applications in a traditional operating system. Add-ons are able to run arbitrary code with the full privileges of the browser, and many useful extensions require potentially dangerous capabilities such as arbitrary file system access and the ability to connect to arbitrary remote servers.

In addition to the complexity of web applications, most browsers also provide the ability for users to add functionality developed by third-parties which modifies the browser's behavior or appearance. There is considerable confusion in terminology between *extensions* and *plugins*. Plugins (§3.3.1) are platform-specific but work across different browsers, while extensions (§3.3.2) are browser-specific but (usually) cross-platform. There are also (in Mozilla terminology) *themes* and *language packs*, which affect only the appearance of a browser and do not affect its underlying behavior.

3.3.1 Plugins

NPAPI

Originally developed as a proprietary API for Netscape Navigator 2.0, the Netscape Plugin API (NPAPI) now describes a standard plugin interface which is implemented by most modern browsers (with the exception of Internet Explorer, which dropped support as of version 6 in 2007). The NPAPI interface is intended for rendering complex media files within web pages, such as audio, video, or PDF documents. NPAPI plugins run as native code on the host operating system; they are instantiated by the browser when it encounters an Internet media type (MIME type) which the user had designated to be handled by the plugin. The plugin code then runs with the full privileges of the browser, receiving a stream of

[32] "Google Removes Suspicious Mobile Apps from Android Market", eWeek Security Hardware & IT Security Software Blog, `http://www.eweek.com/c/a/Security/Google-Removes-Suspicious-Mobile-Apps-from-Android-Market-758811/`

[33] "Android Market Content Policy for Developers", `http://www.android.com/us/developer-content-policy.html`

data from the browser and then rendering that data visually and/or audibly given screen space in the browser.

The API itself is relatively small, the specification maintained by the Mozilla foundation[34] consists of 16 functions implemented by the browser and 25 functions implemented by plugins. However, plugins are notoriously difficult to program and maintain, as they must be implemented separately to run natively on multiple platforms and also may need to interact differently with different browsers. As a result, there are a small number of very popular plugins developed by large software firms, and there is little development of plugins by individuals or smaller groups.

The main mitigation against malicious plugins is discouraging the widespread use of NPAPI. The Mozilla project maintains a list of just 7 recommended plugins which it lists in its add-on directory:[35] Adobe Reader, Adobe Flash Player, Java, Quicktime, RealPlayer, Shockwave, and Windows Media Player. Google does not officially list plugins for Chrome,[36] stating to developers that "NPAPI is a really big hammer that should only be used when no other approach will work."

Despite their few numbers though, NPAPI plug-ins were the single largest source of reported security holes in 2007 [23] with hundreds being documented. Despite this, research into confinement techniques has been limited until recently due to the performance demands of plug-ins which must handle streaming video and audio. Google's Native Client project [44] is an attempt to address this by providing a high-performance sandbox for running native x86 web applications, but is still in an experimental phase.

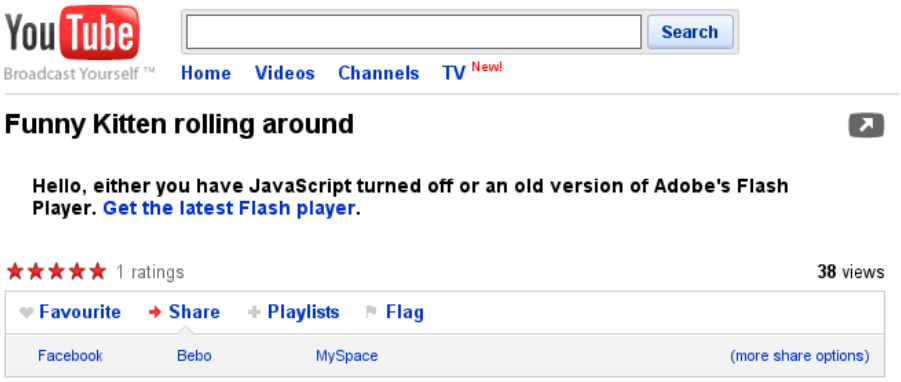


Figure 3.9: Error due to missing Flash Player plugin at `www.youtube.com`

Another common problem is the lack of standardised installation procedures for plugins. While Firefox does have a plugin-finder which will attempt to install plugins from a trusted source to handle media files, many websites will implement their own links to plugins hosted at plugin developer websites, as seen in Figure 3.9. These plugins are often installed by downloading and running an executable installer program directly [34]. As a result, malware has been seen in the wild which tells users they need to download a "new video plugin" which can be really arbitrary malware.[37] Despite the efforts by Mozilla to develop an automatic plugin installer, this social-engineering problem is difficult to fix due to years of user confusion about plugins.

ActiveX

Microsoft Internet Explorer originally supported NPAPI, but phased it out in 2003 in favor of its proprietary ActiveX API which enables much of the same functionality. ActiveX is similar in purpose to NPAPI, although it is much more complicated, also being supported by non-browser Microsoft products such as Office. It is not a single API but rather a marketing term for a series of API's, some of which can be used to implement NPAPI-like plugins for Internet Explorer [13]. Microsoft makes no clear distinction

[34] `https://developer.mozilla.org/en/Gecko_Plugin_API_Reference`
[35] `https://addons.mozilla.org/`
[36] `https://chrome.google.com/extensions`
[37] http://www.avertlabs.com/research/blog/?p=152

to users of the difference between ActiveX controls and browser extensions (see §3.3.2), listing both in its add-on directory for Internet Explorer.[38]

A key technical difference is that ActiveX items can be included in web pages which specify location to fetch the correct ActiveX plugin. In initial implementations, no user intervention was required and ActiveX plugins would be installed automatically. This led to numerous problems with so called "drive-by-downloads" in which malicious websites automatically install plugins which spread malware to the user's machine [34]. Microsoft has responded by progressively increasing the level of user interactivity required for plugin installation, now incorporating dialogue boxes which check for a signature on the required plugin and ask the user if they trust the publisher, as seen in Figure 3.10. Plugins can also be installed by `.exe` installers downloaded from plugin provider websites, and social engineering attacks are equally applicable.

Officially, the Mozilla foundation does not support ActiveX due to its platform-specific nature and security problems.[39] However, it is still possible to use ActiveX controls with Firefox on Windows using a special NPAPI plugin which implements the ActiveX interface.[40]

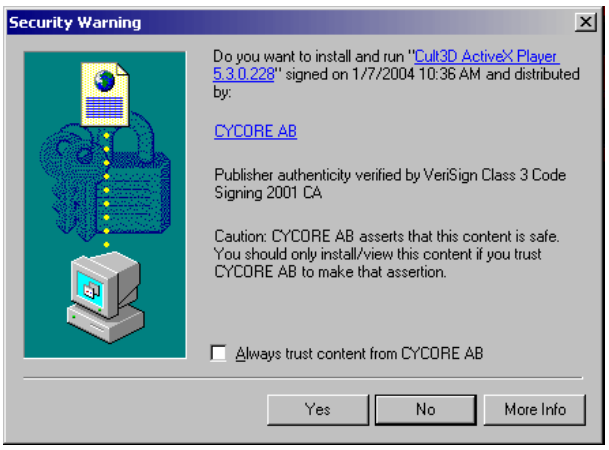


Figure 3.10: ActiveX installation process

3.3.2 Browser Extensions

Mozilla Firefox

Mozilla Firefox, while maintaining roughly half of Microsoft Internet Explorer's market share,[41] boasts the most diverse collection of extensions available for download. This is the result of the browser's initial design philosophy, which aimed to be more lightweight than its predecessor Netscape Navigator by removing all non-essential functionality and creating an easy-to-use development platform for users to optionally add functionality. Mozilla's statistics[42] as of February 2010 show nearly 2 billion extensions have been downloaded from its website alone, with more than 10 million more downloaded per day and approximately 200 million currently in use. Over 10,000 extensions are currently listed, with the most popular, AdBlock Plus, being downloaded over 70 million times.

At a high level, the entire display of the Firefox browser is encoded in Mozilla's XML user-inteface language (XUL),[43] an extension of the standardised document object model (DOM) for representing the contents of a web page. An instance of the browser is the root node of an XUL tree stored in memory, with the various window panes, buttons, menus, being hierarchical elements of the XUL document. The current web page is in fact just an element of the tree.

[38] `http://ieaddons.com/`
[39] `http://support.mozilla.com/en-US/kb/activex`
[40] `http://code.google.com/p/ff-activex-host/`
[41] `http://en.wikipedia.org/wiki/Usage_share_of_web_browsers`
[42] `https://addons.mozilla.org/en-US/statistics`
[43] XUL is also used to render other Mozilla applications such as the Thunderbird email client, which support extensions in a very similar manner.

Developers can extend the standard functionality of the browser by adding content and scripts to the XUL. These extensions are written in primarily in JavaScript, but can also include HTML and CSS for rendering. The browser makes a key distinction between content in the browser's "chrome" (standard interface items and extensions) and "content" (displayed web pages). JavaScript running in "chrome" elements has much greater power, with the ability to read and write to the file system and open its own network connections with full control over the communication protocol. In addition, chrome JavaScript can launch arbitrary code on the host operating system throught the XPCOM API, and many extensions include native code or scripts. An "extension" consists of a series of overlays (or additions) to the standard browser XUL, packaged with any supporting scripts or code accessed through XPCOM, along with any content or media files, packaged in a standardised `.xpi` file which includes an installation manifest file.

Thus, while extensions are designed to be easy to develop by programmers versed in JavaScript, they are not constrained by the normal JavaScript security model. Simple "spy" extensions have been developed by researchers which record all of a user's interaction with the browser and send it to a third-party [28]. This extension can be modified into a full root-kit extension which enables full control of the browser, and indeed the user's machine if the browser has native root privilege. A malicious extension, once installed, can even use its file system access to overwrite the core code of the browser to hide its presence. It is considered necessary to give extensions such power because many useful extensions require it. For example, debugging extensions exist which allow developers to bypass TLS certificate errors or modify submitted form data, while the Alexa toolbar[44] collects traffic data from its users to rank the most popular websites.

Besides intentionally malicious applications, "benign but buggy" applications can pose a major security risk as well. Due to the power provided to extensions and the difficulty of writing secure JavaScript, insecure extensions which interact with untrusted websites may accidentally run untrusted content as JavaScript code, enabling remote attackers to take over the browser.

There are a number of Mozilla-specific JavaScript extensions to help developers limit the security damage from bugs in their code.[45] A key component is the separation between "content" and "chrome" elements in the XUL DOM for the browser. Scripts running in content elements are sandboxed and not allowed to modify or see the chrome elements of the DOM. Extensions which render content (such as sidebars which scroll through news headlines) can mark their displayed elements as "content" frames which sandboxes any scripts from modifying the browser chrome, and can disable JavaScript and plugins within these frames. There is also a standard method `nsIScriptableUnescapeHTML.parseFragment()` is provided to "unescape" untrusted HTML and remove scripts. When an extension needs to run third-party code,[46], a special `evalInSandbox()` call is supported to run untrusted JavaScript. Sandboxed HTTP requests can be made which don't send the user's normal cookies along to prevent accidental privacy leaks.

Still, these sandboxing techniques are purely optional and a study of popular extensions revealed that most carry far more privilege than they actually need [5]. A project to more strictly confine extensions, called Jetpack,[47] has been undertaken, but with the current Firefox architecture Mozilla relies on manual vetting and trust to prevent the distribution of malicious or buggy extensions.

Figure 3.11: Mozilla add-on directory listing with user rating

Mozilla maintains a directory of approved extensions signed by Mozilla and hosted at the Mozilla website.[48] Each extension is displayed along with user ratings and download statistics, as seen in Fig-

[44] `http://www.alexa.com/`
[45] `https://developer.mozilla.org/en/Security_best_practices_in_extensions`
[46] For example, the popular GreaseMonkey extensions enables users to inject their own scripts into pages.
[47] `https://jetpack.mozillalabs.com/`
[48] `https://addons.mozilla.org/`

ure 3.11. While proving any code to be "safe" is undecidable, JavaScript poses a number of technical challenges that make static analysis extremely difficult and useful tools for dynamic analysis of JavaScript extensions are a current topic of research [17]. Thus, a team of volunteer reviewers manually examine all add-ons before they are made public, conducting source-code audits[49] and running a series of automated tests.

The editors also seek to limit buggy extensions by checking for a number of security best practices[50] such as use of JavaScript's `eval()` function with insecure data, or loading scripts from unauthenticated servers. The editor's guidelines state that extensions will blocked for installing any spyware or malware on the user's computer (or for launching any installation program), there are also rules against legally-dubious extensions which aid in online crimes such as click fraud.

Little information is available on the editors' web site[51] about the make-up or size of the volunteer team, only that they are selected from developers who have submitted successful extensions themselves. However, statistics are published[52] about the editors' workload. In 2009, around 1000 reviews per month were conducted, roughly evenly split between new extensions and updates to existing extensions. There are also statistics about the queue of extensions awaiting review which reveal that the median length of review is around 3 weeks.

Users have the ability, however, to install applications not reviewed by the editors. Developers can host unreviewed "beta" extensions on the official add-on website, which users can install by creating an account and stating that they are aware of the risks. Extensions can also be hosted on arbitrary third-party websites, which will automatically trigger[53] the extension installation dialogue as seen in Figure 3.12.

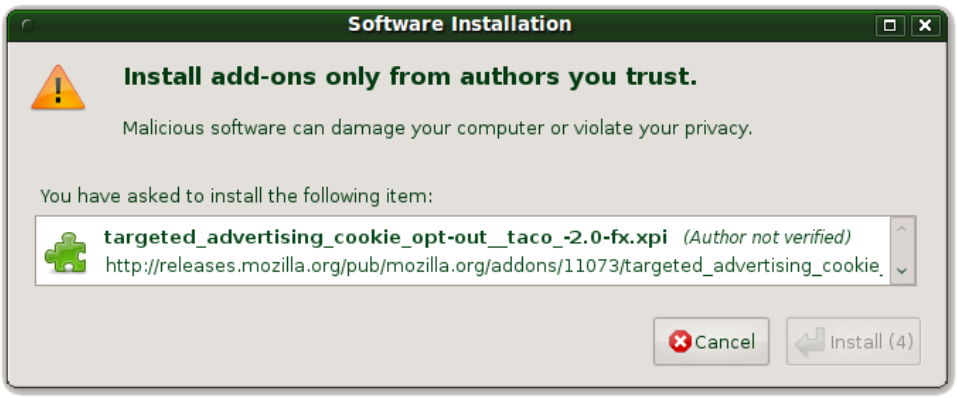


Figure 3.12: Installation dialogue for an untrusted Firefox extension

In addition to normal installation, automatic updates of extensions are supported. These updates go through a similar review process to new extensions.[54] This update process has been shown to be a weakness in the past, as some applications accessed their updates on unauthenticated servers,[55] enabling network attackers to inject malicious updates which could take over a user's browser. Current policy forbids any automatic updates not hosted by Mozilla's own servers, and editors will not approve applications which implement their own update mechanism.

The vast majority of Firefox extensions are provided for free. Mozilla does not forbid the sale of extensions, but it does not allow paid extensions in the official online directory. Other applications are given away for free in the official directory, but link to a premium version which is hosted privately. Still others are ad-supported, or request donations users. Mozilla has recently embraced the donation model

[49] Extensions are required to be submitted with full source code for review, though they can then be distributed with the source code obfuscated.

[50] `https://developer.mozilla.org/en/Security_best_practices_in_extensions`

[51] `https://wiki.mozilla.org/AMO:Editors`

[52] `https://wiki.mozilla.org/AMO:Editors`

[53] Ironically, this is enabled by a lightweight NPAPI plugin (the Default Plugin) that detects that `.xpi` content type and pops-up the browser's extension manager.

[54] except for a small number of trusted "Instant Update" applications which have been approved to submit updates without review

[55] http://paranoia.dubfire.net/2007/05/remote-vulnerability-in-firefox.html

and now includes donation links next to applications listed in the official directory. Still, monetising extensions is very difficult, as there is a large community of hobbyists maintaining the most common extensions for free, and advertising-based extensions tend to be very unpopular [22]. Given the status of the Firefox browser as a free and open-source project, there are also cultural issues as many users are reluctant to pay for additional functionality.

Microsoft Internet Explorer

Microsoft Internet Explorer has supported extensions since version 4.0 in 1997. Explorer is written as an "object" in Microsoft's Component Object Module interface which has been used for most Windows applications since 1993, enabling it to interact with other COM objects implemented as DLL's. Extensions for Explorer are simply DLL's which override one of a number of extension-specific methods and add a specific registry key which notifes Explorer to load them upon startup. These can be installed using the same Windows Installer process for installing applications (§3.1.1) or from within the browser since the release of Explorer version 6.0 in 2004.

Microsoft uses a large number of terms for extensions of various types, such as shortcut menu extensions, accelerators, toolbars, explorer bars, download managers, search providers, and web slices. Each corresponds to a specific type of possible extension which implements a specific COM method called by Explorer. For example, a download manager must implement only the `IDownloadManager::Download()` method, which will be called by Explorer when a user wishes to download a file, allowing the DLL to override the standard download manager. From a security point of view, these differences are not important, as all types of extensions run as native code in the client with the ambient privilege of Explorer, and thus malicious extensions can take over control of the machine.

There are is also a generic browser extension type, called a "browser help object," which is given direct API access to modify any behavior of Explorer by receiving a pointer to its `IWebBrowser2` COM object. This is the most powerful extension interface and thus many extensions choose this interface. It has been used in the wild in several documented cases to create malware extensions which log all of the user's browsing activity, such as software installed by the `Download.ject` worm.[56] Still, malicious browser extensions are not a primary concern, since the browser extension installation process enables an attacker to install an arbitrary DLL file which can override other core aspects of the systems.

A more common problem is semi-legitimate "spyware" extensions which record the user's browsing habits and send them to a remote server. Many such extensions have been written [43], often offering a search toolbar or promising to accelerate web browsing performance.

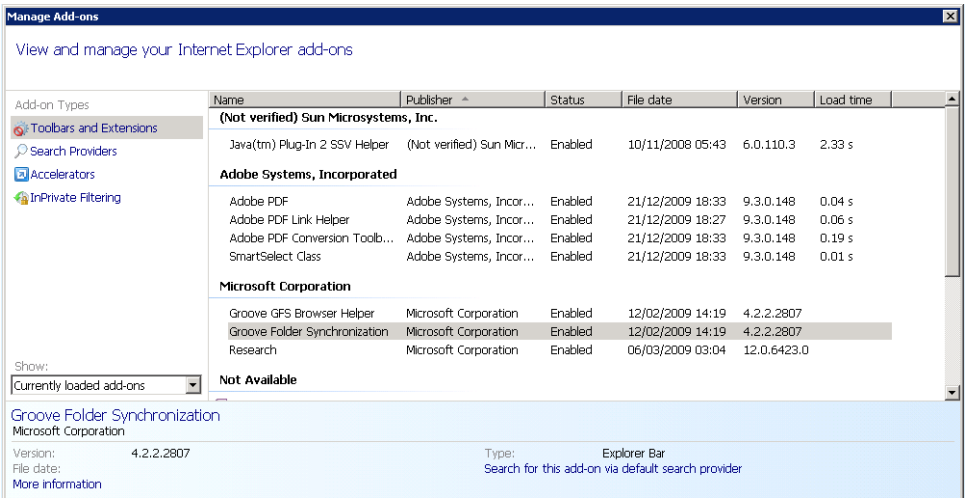


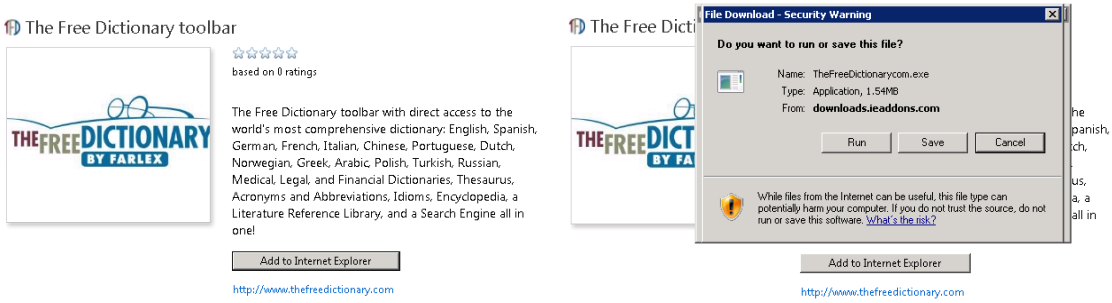
Figure 3.13: Add-on manager in Internet Explorer 8.0

Versions 4.0 and 5.0 of Explorer offered no interface for users to examine what extensions had been added, making it impossible to detect the presence of invisible extensions without examining the registry.

[56] `http://www.us-cert.gov/cas/alerts/SA04-184A.html`

Since version 6.0, Explorer provides a graphical Add-on manager, as shown in Figure 3.13. This gives users the ability to inspect all extensions and disable or enable them Interestingly, it does not allow for deletion of extensions, this must be done through Windows' uninstall facilities.

Unlike Firefox extensions, there does not exist an in-browser mechanism for add-on installation. Microsoft does maintain an online gallery of available applications,[57] but the installation process consists of downloading and running executable installer files, as seen in Figure 3.3.2. A rudimentary ratings system is enabled, but there is no information provided as to how extensions are chosen for listing and what (if any) vetting they have undergone. No download statistics are provided, but the site lists 576 extensions for download, compared to over 10,000 listed Firefox extensions, and the Explorer site receives approximately 5% of the web traffic that the Firefox site receives according to Alexa.[58]



(a) Gallery Entry (b) Download Pop-up

Figure 3.14: The online Explorer add-on gallery

Most add-ons are installed by downloading and running executable install files, making it much it difficult to estimate the market for extensions, though there are paid as well as free extensions available for download. Explorer's help menu also acknowledges that browser extensions are often pre-installed by computer vendors, or are installed as a part of other software installations. Dell, for example, received criticism in 2005 for bundling a toolbar called "My Way Search Assistant" on all new PC's it sold which reported browsing statistics to Dell servers.[59]

The population of extension developers is markedly different from Firefox or Chrome. Due in part to the more complicated development model, there are fewer hobbyist developers and more commercial developers. Paid extensions are far more common for Explorer.

Google Chrome

The extension architecture in Google Chrome, launched only in 2009, is related in many ways by the Firefox architecture, although it has made several major security changes based on observed problems in Firefox [5]. Like Firefox, Chrome extensions are mainly written in the standard web-development languages JavaScript, HTML, and CSS and are normally cross-platform. Extensions are able to interact arbitrarily with the DOM of loaded pages. Unlike Firefox, the current Chrome architecture does not give extensions the ability to override arbitrary aspects of the browser's display, instead implementing a fixed number of user interface elements that extensions can add such as menus and status bars.

The Chrome extension system is designed to have extensions run with the least necessary amount of privilege. Chrome extensions must declare in their manifest file all browser elements that they request rights to interact with. For example, an extension which needs to interact with the browser's tabbing behavior needs to add the line `"permissions":["tabs"]` to its manifest file. The manifest file also requires specifying all remote websites an extension may need to interact with. While it is possible to request unlimited network access, site-specific extensions can request to be limited to one domain, only running when a page from that domain is loaded and only being able to communicate over the network to that domain.

[57] `http://www.ieaddons.com`
[58] `http://www.alexa.com`
[59] `http://www.theregister.co.uk/2005/07/15/dell_my_way_controversy/`

Chrome has also taken several steps to limit potential damage from buggy applications by separating "content" scripts which can interact with untrusted web content from "core" scripts which can affect UI elements and have extended privileges. These two scripts can only interact via a narrowly-defined message passing interface. Content scripts are further restricted in their interaction with untrusted website scripts through an "isolated world" implementation of the DOM, in which only displayed DOM elements are shared, but extension scripts and website scripts cannot access the same variables. Running native code is supported for Chrome extensions through the NPAPI interface, and native code is run in a separate process. Malicious applications which include native code can still seize control of a user's machine, and extensions without native code can still take over the browser and steal any of a user's sensitive data.

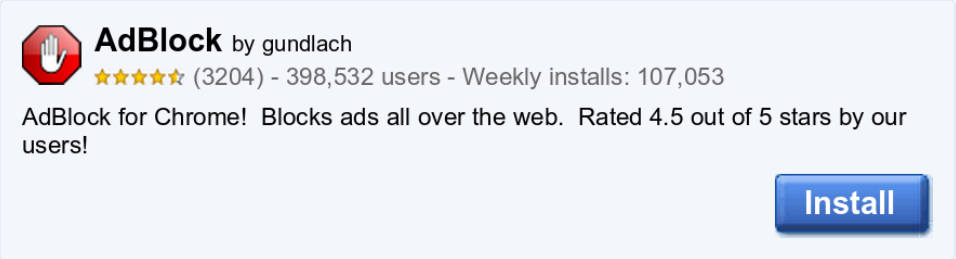
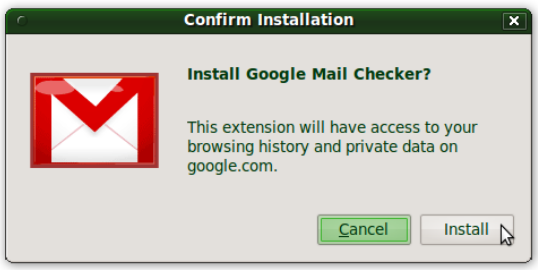
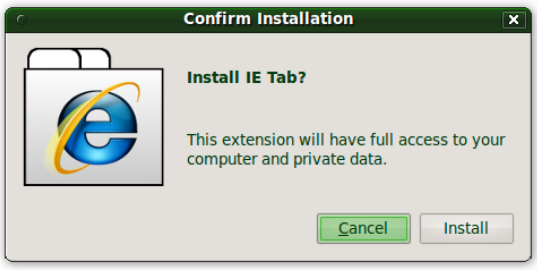


Figure 3.15: Chrome extension online listing

While Chrome has stricter technical safeguards, it has laxer vetting procedures. An official directory of Chrome extensions is maintained online, as seen in Figure 3.15, but these applications are not reviewed unless they include native code. User reviews are posted, along with download statistics. Payment and donation are not supported on the official site, though applications are given a larger amount of space for self-description and some developers post links requesting donations. The growth of the Chrome extension directory has been very impressive, with over 3,000 applications listed in the first 3 months, and the most popular, AdBlock, having 400,000 downloads. Developers are also free to host extensions off-site, and are allowed to charge money for extensions, though this is relatively rare.



(a) Lower-privilege extension



(b) Higher-privilege extension

Figure 3.16: Google Chrome extension installation dialogue

Chrome supports installation from within the browser whenever a `.crz` content type is seen. A dialogue box pops up (Figure 3.3.2) which requests the user's consent. Chrome scans the manifest file for requested permissions gives the user a summary of what is being requested. For example, the extension in Figure 3.16(a) is specific to one website, while the extension in Figure 3.16(b) includes full access to the browser and native code. Also unlike the Firefox UI, there is no indication of the source of the extension and no facility for signed code. Finally, Chrome offers an extension management UI (Figure 3.17) which enables viewing, disabling, and uninstalling extensions. Automatic update is also supported in Chrome,

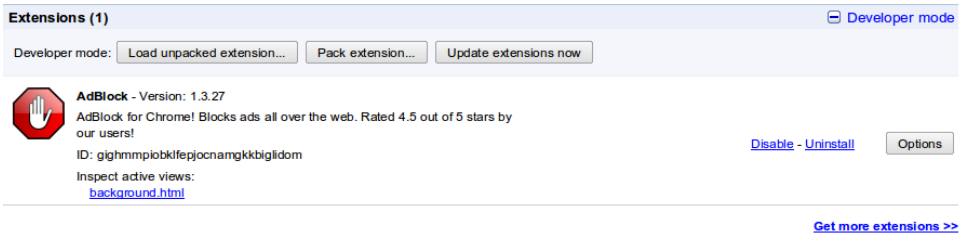


Figure 3.17: Extensions manager in Chrome

3.4 Social Networks

3.4.1 Facebook

As of writing, the social network Facebook has over 400 million users.[60] Facebook allows developers to create third-party applications to access user information. As the most popular social networking platform in the world, it has been targeted by many applications which extract users' personal information *en masse* [20, 4, 9, 10]. Facebook has both more and less effective access controls than traditional operating systems: it is possible to constrain what information is shared with applications, but it is impossible to even observe what they do with that information once they have it.

By default, Facebook applications can access any information which is available to Everyone in the user's privacy settings, as well as "publicly available information," which Facebook defines to include "name, Profile picture, gender, current town/city, networks, friend list and pages," some of which is not displayed to logged-in Facebook users. In addition, applications can request access to more personal information if they require it; such requests must be approved by the user.[61] Finally, users can share information about their friends with applications; restrictions on this behaviour are shown in Figure 3.18.

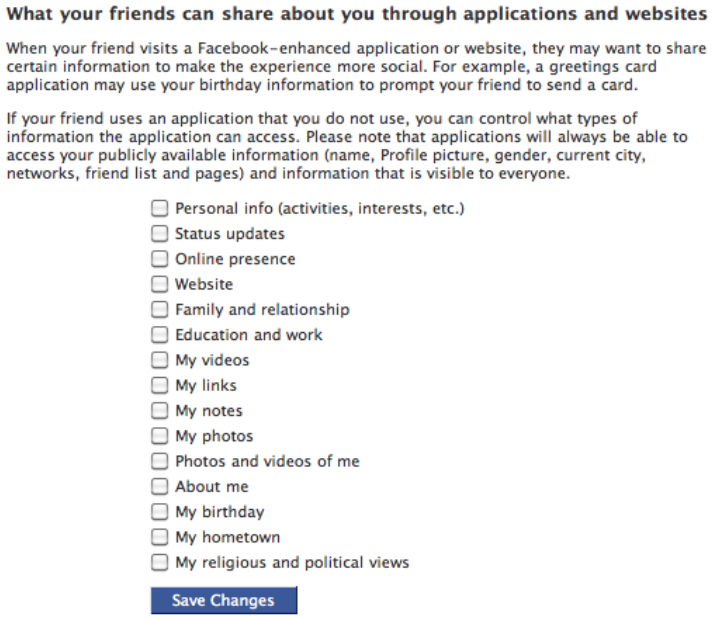


Figure 3.18: What a user's friends can share with a Facebook application.

Once applications have been granted personal information, they can do anything with it. Facebook applications have been shown to leak personal information intentionally [26] and in direct violation of the

[60] `http://www.facebook.com/press/info.php?statistics`
[61] `http://www.facebook.com/settings/?tab=privacy&section=applications&field=learn`

Facebook Platform Guidelines.[62] In some cases, developers would release Facebook applications such as games for the purpose of collecting user information and sending them targeted spam or premium text messages, and for some time Facebook's unofficial policy was "stepping in only when the violations were so egregious that [their] call center was getting flooded with complaints."[63]

Even applications which do not appear to be malicious have leaked vast quantities of personal information: one Facebook blogger sought, and found, one cross-site scripting (XSS) bug in a Facebook application every day for 30 days.[64]

Methods of improving security have been proposed but ignored. One method is proxying user information: many applications use a user's name, but do not actually need to know what that name is. Rather than granting blanket access, Facebook could require applications to insert a field such as `$name` wherever personal information is to be used, and Facebook could populate it without revealing any personal information to the application [21].

Facebook attempted to run a certified application scheme, but it failed to detect obvious privacy violations in applications,[65] and after a few months, it was cancelled.[66]

3.4.2 OpenSocial

OpenSocial (`http://www.opensocial.org/`) provides APIs for collaboration among social *applications*, *containers* (social networking websites) and *clients* (web browsers). The goal is to allow social applications to be written once and run on many containers, such as LinkedIn, MySpace, NetLog and orkut.[67] Positioned as an open alternative to proprietary application platforms such as Facebook, OpenSocial provides similar functionality, including similar privacy concerns.

OpenSocial provides the ability to write three types of social apps:[68]

1. *Social mashups*, in which gadgets run inside the user's Web browser and request personal information from the containing social network,
2. *Social applications*, which rely on Facebook-style external servers for rendering and
3. *Social websites / social mobile applications*, which are external websites that request personal information from a social network.

In all cases, external servers can have access to personal information; even social mashups can make requests from remote sites. Direct requests are prevented by the web browser's Single Origin policy, but requests can be proxied by the social network itself.[69] Once personal information has been sent to an external server, no enforcement of information flow can be provided; so all constraints must be imposed at the app / container interface. This access control is entirely at the discretion of the OpenSocial container, and can be even coarser than that provided by Facebook.

Figure 3.19 shows the very coarse access control provided by one OpenSocial container, Google's Orkut. In this case, users have just four constraints which can be imposed on installed applications: not allowing an application to appear on the user's profile page, not allowing it to communicate with friends, requiring previews before such posting occurs and restricting access to photos. In all cases, applications only have access to the user's public profile information, not that which has been restricted to e.g. the user's friends.[70]

[62] `http://www.lightbluetouchpaper.org/2009/06/09/how-privacy-fails-the-facebook-applications-debacle/`

[63] `http://techcrunch.com/2009/11/01/how-to-spam-facebook-like-a-pro-an-insiders-confession/`

[64] "The Month of Facebook Bugs Report", Social Hacking, `http://theharmonyguy.com/2009/10/09/the-month-of-facebook-bugs-report/`

[65] `http://theharmonyguy.com/2009/05/28/about-that-verification/`

[66] `http://www.allfacebook.com/2009/11/facebook-verified-apps-end/`

[67] `http://wiki.opensocial.org/index.php?title=Containers`

[68] "OpenSocial Developer's Overview (v0.9)", `http://wiki.opensocial.org/index.php?title=OpenSocial_Developer's_Overview_(v0.9)`

[69] "Remote Data Requests Developer's Guide (v0.9)", `http://wiki.opensocial.org/index.php?title=Remote_Data_Requests_Developer's_Guide`

[70] "Terms of Use of Applications", `http://www.orkut.com/AppTerms.aspx`

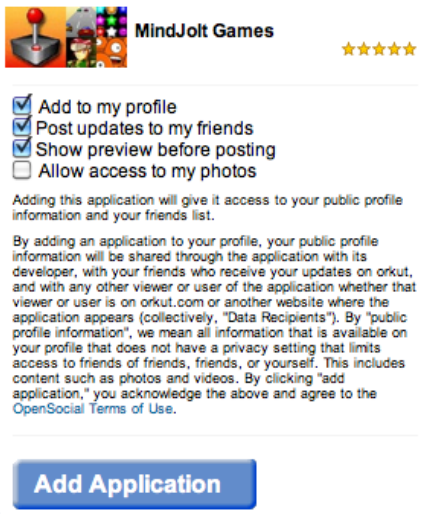


Figure 3.19: Adding an OpenSocial application to Orkut.

Vetting

OpenSocial applications are made available through per-container directories, e.g. the LinkedIn Application Directory (`http://www.linkedin.com/apps`). Each container is free to vet applications, and some do: NetLog applications are moderated for their use of links, localisation, logos, etc.[71]

In some cases, however, applications can be added from arbitrary URLs. Orkut is an example of a container which allows applications to be added in this way, as shown in Figure 3.20. If users install applications in this way, the container can take no responsibility for the application.

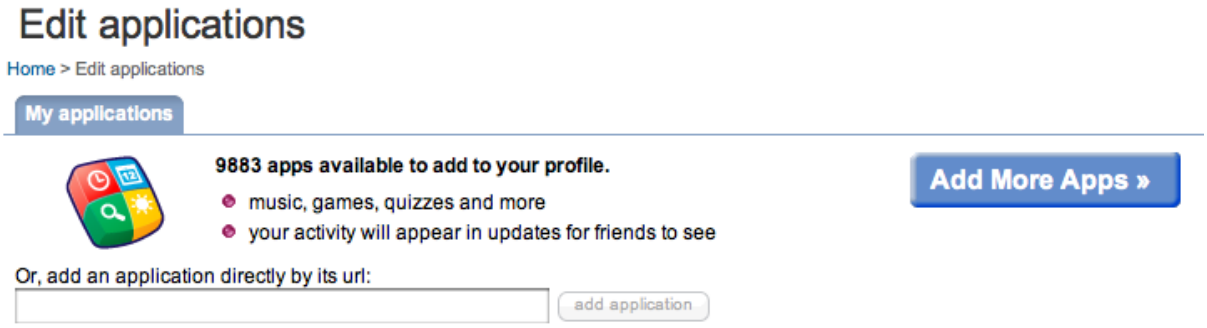


Figure 3.20: Orkut's "Edit Applications" page.

Summary

We have found considerable variation in user and system protection among the four categories of application platforms (traditional operating systems, mobile phones, web browsers and social networks) that we have considered. We have also found variation within the categories, but far less than that which exists between them: it seems that the differences between markets are far more significant than the differences within markets.

[71] "Application Guidelines", NetLog Developer Pages, `http://en.netlog.com/go/developer/documentation/article=appguidelines`

Chapter 4

Analysis

We have examined case studies of ten computing platforms and teased out the economic roles common to each platform's application marketplace. We now provide a summary of each market, an economic analysis of the security problems that we have found and suggestions for future improvement.

4.1 Current Markets

Figure 4.1 shows the design space for these case studies, formed by the axes of user and system protection. Each case study's subject is located according to how much protection is afforded to both users and the system itself, graded on a subjective scale from 0 (no protection) to 100% (complete protection).

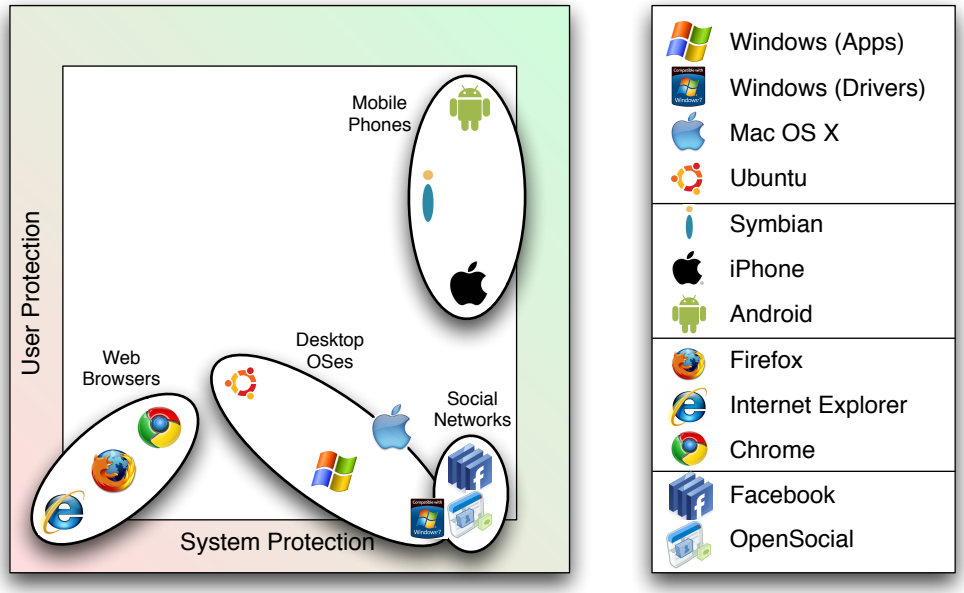


Figure 4.1: Case studies in the user and system protection spectra.

4.1.1 Traditional Operating Systems

Traditional operating systems take some care to protect themselves from modification by malicious applications at runtime, but software installers are generally opaque and highly privileged, allowing users to get more than they may have bargained for. Once installed, applications operate with ambient user authority, providing malicious applications with the same level of access to information as users. Furthermore, application vetting is impossible both before installation [27] and at runtime [41].

Windows (Apps) Windows protects itself from malicious installer packages via WFP and malicious or compromised applications via WFP and a Biba-like MAC policy. Users are afforded almost no protection, however, so we assign it a $(system, user)$ protection rating of (0.6,0.1).

Windows (Drivers) Owing to Microsoft's work in driver verification and certification, Windows-based computers are much more reliable today than before this work began. This increased system reliability contributes to a more positive user experience than might otherwise be obtained, but it also makes it more difficult for malware to install itself using the device driver framework. We assign the Windows driver framework a system protection rating of 0.8, and as it is not directly related to user protection, we assign a user protection rating of 0.

Mac OS X Mac OS X provides a commonly-used installation method that does not require system privilege, but the other commonly-used method does, and this method allows arbitrary installer binaries to be executed with privilege and possibly install unwanted software. Once installed, applications have ambient user authority and can register themselves Thus, we assign a protection rating of (0.7,0.2).

Ubuntu Ubuntu has a package management mechanism in place which could protect against many system attacks, but it allows arbitrary installer script execution with system privilege, and these scripts use no confinement mechanism. Ubuntu-supplied applications and their packaging scripts can be audited, and those with the power to create them do undergo a selection process that might weed out some malware authors. We assign Ubuntu a protection rating of (0.4,0.3).

4.1.2 Mobile Phones

Mobile phone platforms provide much stronger user and system protection than their desktop counterparts. Data and state are protected by a combination of technical confinement, application certification and user permission. Those platforms which are weak in one area tend to be stronger in others, so the field is relatively level.

Symbian Symbian OS 9.1 and later provide strong controls on both reading and writing user and system data. The pre-defined permissions model is both coarse and inflexible, but it does provide remarkably good protection when compared to other operating systems of the time (i.e. desktop OSes). The introduction of certification reduced malware incidence dramatically, but also keeps some decisions about user data out of the hands of users. We assign a protection rating of (0.8,0.7).

iPhone The iPhone protects the system very effectively from malicious applications, and it also protects applications from each other. Protection of user information, however, relies less on technical measures than application vetting, which has been foiled in the past. We assign the iPhone a protection rating of (0.9,0.5).

Android Android, with its DAC- rather than MAC-based security mechanisms, provides less assurance than the iPhone of system protection, but the actual protections are likely as effective. Protection of user information, however, is far more fine-grained than either Symbian or iPhone OS; it seems to have been a priority during design. We assign Android a protection rating of (0.9,0.9).

4.1.3 Web Browsers

Whereas traditional operating systems have all made the leap from co-operating to pre-emptive multitasking, web browsers are largely still monolithic applications whose extensions share their address space and can modify state at will. Google Chrome has started to push browsers towards a model in which extensions declare the permissions they require and run in a separate process, but this will not yield security improvements until asking for all permission makes an application stand out suspiciously and extension processes become extension sandboxes.

Firefox Firefox gives developers the most freedom to change functionality of the browser, and implements security mostly through vetting and trust. We assign Firefox a rating of (0.1,0.1) for its lack of technical protections and review process of unknown quality.

Chrome Compared to Firefox, Chrome's architecture is slightly more limited in giving developers freedom to change core browser functionality, and relies more heavily on technical sandboxing and user choice than on vetting to enforce security. We assign Chrome a rating of (0.2,0.2) for its steps towards the least-privilege execution of extensions.

Internet Explorer Internet Explorer has the most complicated development model and the least well-developed distribution channel, as well as having minimal security apparatus in place. We assign IE a protection rating of (0,0) for providing not protections for either browser or user.

4.1.4 Social Networks

The social network itself is well protected from applications, partially because of the "air gap" that exists between them. An example of the differing interests and understanding of users and operators [3], this air gap is a boon to the system but the bane of users. Once information has been released to an application, users have absolutely no control over its flow, and the social network often provides ineffective or well-hidden protections.

Facebook The Facebook platform itself is well protected from malicious applications, but its design ensures that the flow user information cannot be controlled or even monitored once released. Facebook has provided technical measures to prevent some information from flowing to applications, but other pieces of information cannot be hidden. Furthermore, Facebook has not clamped down on, and perhaps even tacitly approved of, abusive applications like spam. We assign the Facebook Applications Platform a protection rating of (0.9,0.1).

OpenSocial OpenSocial suffers from the same fundamental design flaw as Facebook: applications run on remote servers. In the OpenSocial case, the assurances given at the container/application interface are even weaker than those given by Facebook; user information is even more vulnerable to malicious applications. We assign OpenSocial a protection rating of (0.9,0).

4.2 Economic Problems

Figure 4.1 showed significant clustering of application platforms, demonstrating that inter-market differences in security affordances are more significant than intra-market ones. This suggests that important economic factors are at work.

4.2.1 Negative Externalities

When users of Microsoft Windows pre-2000 experienced "DLL hell," they were actually experiencing negative externalities which led to a tragedy of the commons. Application writers would design their software to overwrite core operating system components because it was easy (and inexpensive) to design, and the costs would be borne by other applications on the system—a negative externality. As more and more applications followed this route, overall system reliability was measurably reduced to the point that Microsoft stepped in to push costs back on to application writers, forcing them to assume more of the true cost of reliability. Unfortunately for users, no such discipline has yet been imposed on the writers of e.g. Internet Explorer extensions: a "browser help object" can modify any IE behaviour, providing a direct benefit to the Developer and spreading runtime compatibility costs over all extensions (as well as the User, in the case of spyware).

Facebook does not charge its users a subscription: its income comes from advertising, sometimes indirectly through companies which leak user information through their *own* advertising platforms and then advertise themselves on Facebook [26]. Users might prefer that applications operate according to the Principle of Least Authority, but the lack of incentive alignment between Facebook and its users

leads to the current situation: using the Facebook Application Platform results in personally identifiable information being leaked by unscrupulous applications and the negligent platform which hosts them.

4.2.2 Asymmetric Information

One primary reason that applications can cause so much trouble is asymmetric information. It is easy to hide functionality inside a Mac OS binary or OpenSocial application server, so only the author knows what the application is designed to do. This is why certifying applications, though popular, is ineffective.

Microsoft certifies device drivers, and both Symbian and Apple have a mobile application certification program, although the latter two have approved applications which did not meet their standards. In the social network space, NetLog subjects applications to a limited approval process that does not test safety as much as branding, and Facebook cancelled its Verified Application program after only a few months. The Certifier can also be an external entity: high-assurance computing platforms have traditionally been associated with Orange Book or Common Criteria certification, although problems with the Common Criteria methodology have yielded spectacular failures [18].

Adverse Selection Information asymmetry has led to adverse selection in some attempts to certify applications as "safe." This has been demonstrated in TRUSTe certification: only the website being certified knows if it is malicious or not, and malicious websites have more incentive to acquire certification than non-malicious ones, which leads to the stunning conclusion that having a TRUSTe certification actually *increases* the probability that an unknown website is malicious [19].

Lemons Market Asymmetric information can be the cause of "lemons markets," but in the case of malicious applications, asymmetric information may be one reason that a lemons market has *not* developed. It is very difficult for a buyer to assess the reliability of a used car, but it is very easy to tell when it has broken down. Malicious applications are very difficult or, in the social network case, almost impossible for the User to detect, and they remain difficult to detect even while they are stealing personal information or sending spam. If the User knew that these activities were occurring, they might lose confidence in the applications market; as it is, she often does not even know that something is wrong.

4.2.3 Moral Hazard

Another classic economic problem that can be observed in the applications marketplace is moral hazard. Moral hazard can easily be observed in the actions of the Distributor and Platform Vendor: the Android Market does not vet applications because the costs of application misbehaviour fall on the User or Administrator, not the Distributor. Similarly, Facebook has little incentive to design and implement a least-privilege operation mode for applications because misbehaving applications violate the privacy of the User, not the availability of the system.

In fact, Facebook's primary goal is growth, and since a social network (like other platforms) is a two-sided market, Facebook has a very strong incentive to make the Application Platform as compelling as possible for the Developer. This means exposing as much personal information as the User (occasionally represented by the Privacy Commissioner of Canada[1][2]) will let them get away with.

Moral hazard also explains why the Developer cuts corners or even intentionally inserts code which acts against the interest of the User: the costs of application code misbehaving are borne by the User or Administrator, rarely by the Developer. The Developer's incentive of getting the code to market quickly or subverting the User's interests for the Developer's gain are not countered by the risk of bearing some of the User's costs.

[1] "Facebook needs to improve privacy practices, investigation finds", Office of the Privacy Commissioner of Canada, `http://www.priv.gc.ca/media/nr-c/2009/nr-c_090716_e.cfm`

[2] "Privacy Commissioner launches new Facebook probe", Office of the Privacy Commissioner of Canada, `http://www.priv.gc.ca/media/nr-c/2010/nr-c_100127_e.cfm`

4.3 Economic Solutions

The economic measures that we propose to improve current application confinement practices can be viewed through the lens of the three broad security economics solutions proposed by Schneier [37]: increased liability, insurance and risk-reducing technical mechanisms. The idea that liability, insurance and technical measures have important roles to play in a security context is not new, but the technical and economic differences between the application marketplace and other contexts is illuminating. In particular, we are heartened to see limited, voluntary adoption of technical protections in some marketplaces, but more remains to be done if the incentives of the various actors are to be aligned.

4.3.1 Liability

One category of economic solution to the problems of moral hazard and the tragedy of the commons is the imposition of software liability on the Developer and/or Distributor. Such imposition has traditionally been discussed in terms of "hard" liability, whether imposed by regulation or agreed to by contract, but a new phenomenon is emerging whereby corporations willingly assume some level of informal, "soft" liability to their reputation, future business, etc. in exchange for increased brand exposure.

Regulation

Liability has traditionally been the province of regulation, and the question has been asked before [37], "why isn't defective software treated like defective tires in the eyes of the law?" Such an approach has been taken, in a limited sense, towards products (which may include software) that endanger safety: EU product liability legislation provides a legal remedy for consumers injured by defective "movables" which cannot be disclaimed by contract [1]. Faults in application software, however, do not often fall into this category: malware may steal credit card details, but it is unlikely to set the house on fire.

Governments could impose "hard" software liability via legislation that holds the Developer responsible for software faults, but such an approach would introduce a "chilling effect" on development. Much of the PC and Internet technology that we enjoy today only exists because it was initially incomplete, giving users a platform to develop on rather than a complete product [46]. Software liability could quash this type of innovation, destroying the next technological revolution before it is even conceived.

Contract

"Hard" liability can also be negotiated by two parties and recorded in contract. Liability may be explicitly acknowledged in contract, but not necessarily disclaimed: in *St Albans City and District Council v International Computers Limited* (in which a defect in ICL software cost the St Albans Council approximately £1M), UK courts found that ICL could not limit their liability to the £100,000 that their standard terms allowed, owing to protections against unfair contract terms which are normally only afforded to consumers [29, 42].

Two businesses which enter into a contract related to the provision of software could likewise negotiate which party will be deemed liable for software faults. Contracts between a business and a consumer, however, are more complicated: most software is accompanied by a license agreement that does not even warrant that the software is fit for any purpose, but consumers are often protected from contract terms which the courts deem unfair. In between these two extremes, it is difficult for either party to know how a contract will later be interpreted by the courts. Thus, we find that effective liability is "soft" liability.

Branding

An implicit liability shift happens when the Platform Vendor's brand is tied to the quality of applications available for the platform. For instance, Microsoft places a much heavier verification burden on driver developers than it does for application developers because driver developers have the power to make Windows crash, which negatively impacts on Microsoft's brand. In the mobile phone space, Symbian introduced the Platform Security project in an effort to be *the* brand that stood for safe, reliable, malware-free operation. Similarly, the Mozilla foundation has spawned the Jetpack project to ensure that Firefox does not acquire an unsavoury reputation as a hotbed of malware.

None of these efforts come with the strong guarantees of a regulatory regime: when Symbian signed malware, no User was entitled to mandatory compensation. Nonetheless, the Platform Security project has resulted in a dramatic reduction in the prevalence of malware on mobile phones, surely a good outcome from the User's perspective.

In all of these cases, brand polishing has occurred after some initial tarnishing. The open question is, can incentives be provided which will make the Platform Vendor in traditional operating system and social network markets guard their brands more jealously? Some markets (e.g. mobile phones) have moved quickly from inception to problem to mitigation; why not traditional operating systems?

Market Power

Another reason for a Platform Vendor to accept increased liability is in exchange for market power as a Distributor. The iPhone App Store applies a more stringent vetting process than the Android Market because Apple has a monopoly on distribution. Apple has an incentive to protect its iPhone brand, but it also has a financial incentive not to employ a testing team. If Apple were only one of many application Distributors, the costs of verification might be balked at, but because Apple has positioned itself as a price maker rather than a price taker, it can afford to do this verification for the sake of its platform.

This implicit liability shift is also weaker than a regulatory shift, but its effect — highly-constrained applications — is a useful one. The technology used to confine applications, Sandbox, has also been introduced into the traditional operating system Mac OS X, where it has been adopted by Google Chrome for some of its internal confinement. Increased market power must be closely monitored but, when it leads vendors to accept liability, it should be encouraged.

4.3.2 Insurance

One way for the Owner, Administrator and User of applications to reduce their risk is to purchase insurance. Today's Certifier essentially says, "I think that application X will not do harmful thing Y." An insurer would provide the stronger guarantee, "if application X does harmful thing Y, I will compensate you with Z." This idea has been explored before with the notion of cyberinsurance [37, 24].

The difficulty in providing insurance in this field is the asymmetric nature of the information provided about applications in many environments, which can lead to adverse selection and moral hazard. The insurer's traditional response is either to provide incomplete coverage or to make observation of the customer's behaviour a condition of coverage [40]. Limiting coverage means that the Administrator's goal of shifting risk has not been achieved; we will thus consider the other response, observation.

Observation of software behaviour is often a very difficult problem, but some platforms have implemented effective confinement mechanisms. Paid insurers with a desire to observe software behaviour could provide the needed incentives for the Platform Vendor to build least-privilege operation and transparent installers into his platform. Thus, a trend towards insurers should be encouraged through liability.

4.3.3 Signalling

Another means of reducing risk for Administrators (or insurers) is to restrict software usage to those applications which can engage in signalling. Current certification schemes are a form of signalling, although they are not always effective, and misunderstandings about evaluation and certification can lead to spectacular failures [18]. Signals may not prove safety or reliability, but if they work correctly, they can overcome some information asymmetry: a system which has been certified to Common Criteria EAL5 is less likely to be malicious than one selected at random. Improved signalling mechanisms would include technical measures such as package constraints, application sandboxing and code review.

Package Constraints

In §3.1.3, we showed that the Debian installation mechanism is insufficient to prevent system tampering by pre/post-install scripts. Debian packages without such scripts, however, do have strong confinement properties, and it is a much simpler matter to prove the lack of scripts than to prove the safety of scripts which do exist. Similarly, Mac OS X Installer packages can easily be shown not to contain installer binaries which run with privilege, and applications can also use the *manual install* method ("drag and drop"), which does not require system privilege.

Packages which impose constraints on themselves in this way signal nothing about their run-time properties, but they do break through the information asymmetry that is normally inherent in installers.

Application Sandboxing

Applications such as Google Chrome use self-compartmentalisation to confine the execution of untrusted Web sites. This approach to security provides no guarantee about Google's own code, but it does speak to a concern for security which required significant effort to implement, and which would make Google malice easier to detect. We can, thus, consider it a signal which provides some information to counter the asymmetry that otherwise exists.

Application sandboxing would also reduce certification costs tremendously. One difficulty associated with certification schemes is that they can be tremendously expensive and time-consuming if they are to provide high levels of assurance. Review of the design and implementation process, as well as modifications, can often take over a year of expensive consulting work. Examining opaque application binaries for obvious faults is far more expeditious—and perhaps likely to generate repeat business—but it also provides less assurance and can lead to revoked certifications. Certifying applications which run within a set of known constraints is much more tractable than the general problem of identifying malice. This would reduce the cost of many existing, low-assurance schemes (arguably to their actual value), but it could create a market for affordable high-assurance certification.

Code Review

Another signal which applications can employ is code review, opening application source code for inspection. A piece of software such as Linux or Windows is far too complex for any individual to audit completely, but the fact that their sources are available (in the latter case, under a special license to academic or industrial partners) makes malice riskier to insert, and makes openness to code review—and assurance that the code being reviewed is the code being run—an important security signal.

Reputation

Finally, applications can send signals in the form of reputation. This idea has been explored in social networks, where applications were presented to users along with a rating given by other users [6]. The idea behind this work is that applications which are observed doing things that users do not like (e.g. sharing their e-mail address with third-party advertisers) will receive poor ratings, and that other users can choose not to install such applications. In practice, the fact that a widely-used application has not been observed acting maliciously by non-expert users is not proof that it is not actually malicious, but having a reputation to lose does make malicious behaviour risky, especially if the population of expert users is likely to detect it and other users can verify the detection or at least trust the experts.

Chapter 5

Conclusions

We have examined case studies of application platforms from four categories: traditional operating systems, mobile phones, web browsers and social networks. We have found that the protections afforded users and the systems themselves vary within categories, but more dramatically between categories. Traditional operating systems are best at protecting themselves from malicious applications; users find little or no protection against malicious applications attacking their personal information. Operating systems for mobile phones tend to provide much stronger protection of user information, although malware capable of reading user data has found its way into consumers' pockets. Web browsers are the operating systems which have not yet matured into the lessons supplied by decades of operating systems research and practice. Change appears to be on the horizon, but for now, browser extensions seem largely unlimited in their power to damage users and systems. Finally, social networks demonstrate the double-edged sword that is the air gap: social network systems themselves are well isolated from applications, but so are any controls which might protect user information that has been shared with those applications.

We have also extracted from these case studies eight roles which actors in the applications market assume: the User, Administrator, Owner, Platform Vendor, Distributor, Certifier, Packager and Developer. Each has costs associated with ensuring information protection, or with the failure of that protection. Many of them have the ability to reduce their costs by imposing costs on others.

Finally, we consider the economic phenomena which occur within this market structure and influence the security of application platforms, and we propose economic solutions to problems faced in markets struggling with security. Phenomena of interest include the tragedy of the commons, asymmetric information, moral hazard and two-sided markets. One possible solutions is the imposition of software liability or, preferably, the voluntary assumption of liability through brand protection or the pursuit of market power. Another solution for risk management, whose demand would be increased by increased liability, is a more mature software insurance infrastructure. A final solution that we consider, which would aid and thus be encouraged by software insurance, is technical signalling to overcome information asymmetry. This signalling could take the form of self-imposed software packaging constraints, self-imposed application sandboxing, openness to code review and the employment of a reputation system for applications.

Acknowledgements

We would like to thank Ross Anderson and Robert Watson for most helpful discussion and collaboration on topics of economics and technical application constraints.

Bibliography

[1] *Council Directive (EC) 85/374/EEC of 25 July 1985 on the approximation of the laws, regulations and administrative provisions of the Member States concerning liability for defective products.*

[2] Anderson, J., Diaz, C., Bonneau, J., and Stajano, F. Privacy-Enabling Social Networking Over Untrusted Networks. In *Proceedings of the Second ACM SIGCOMM Workshop on Social Network Systems (WOSN '09)* (2009), pp. 1–6.

[3] Anderson, J., and Stajano, F. Not That Kind of Friend: Misleading Divergences Between Online Social Networks and Real-World Social Protocols. In *Proceedings of the Seventeenth International Workshop on Security Protocols (SPW '09)* (2009), pp. 1–6.

[4] Athanasopoulos, E., Makridakis, A., Antonatos, D. A. S., Ioannidis, S., Anagnostakis, K. G., and Markatos, E. P. Antisocial Networks: Turning a Social Network into a Botnet. In *Proceedings of the 11th Information Security Conference* (2008), Springer, pp. 146–160.

[5] Barth, A., Felt, A. P., Saxena, P., and Boodman, A. Protecting browsers from extension vulnerabilities. In *Proceedings of the 17th Network and Distributed System Security Symposium* (2009).

[6] Besmer, A., Lipford, H. R., Shehab, M., and Cheek, G. Social Applications: Exploring A More Secure Framework. In *Proceedings of the Symposium on Usable Privacy and Security (SOUPS)* (2009).

[7] Biba, K. Integrity Considerations for Secure Computer Systems. Tech. Rep. MTR-3153, MITRE Corporation, 1975.

[8] Bittau, A., Marchenko, P., Handley, M., and Karp, B. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2008), pp. 309–322.

[9] Bonneau, J., Anderson, J., Anderson, R., and Stajano, F. Eight Friends Are Enough: Social Graph Approximation via Public Listings. In *Proceedings of the Second ACM EuroSys Workshop on Social Network Systems (SNS '09)* (2009).

[10] Bonneau, J., Anderson, J., and Danezis, G. Prying Data Out of a Social Network. In *Proceedings of the 2009 International Conference on Advances in Social Network Analysis and Mining* (2009).

[11] Brumley, D., and Song, D. Privtrans: Automatically Partitioning Programs for Privilege Separation. In *Proceedings of the 13th USENIX Security Symposium* (2004).

[12] Chadwick, E. Results of different principles of legislation and administration in europe; of competition for the field, as compared with competition within the field, of service. *Journal of the Statistical Society of London 22*, 3 (1859), 381–420.

[13] Chappell, D., and Linthicum, D. S. ActiveX Demystified. *Byte 22*, 9 (1997), 52–64.

[14] Cook, B., Podelski, A., and Rybalchenko, A. Termination proofs for systems code. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation* (2006).

[15] Cook, S. A. The complexity of theorem-proving procedures. *Proceedings of the Third Annual ACM Symposium on the Theory of Computing* (1971), 151–158.

[16] Dennis, J., and Horn, E. Programming Semantics for Multiprogrammed Computations. *Communications of the ACM 9*, 3 (1966).

[17] Dhawan, M., and Ganapathy, V. Analyzing Information Flow in JavaScript-Based Browser Extensions. Tech. Rep. 648, Rutgers University, 2009.

[18] Drimer, S., Murdoch, S., and Anderson, R. Thinking Inside the Box: System-Level Failures of Tamper Proofing. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy* (2008).

[19] Edelman, B. Adverse Selection in Online "Trust" Certifications. In *Proceedings of the Workshop on the Economics of Information Security (WEIS)* (2006).

[20] Felt, A. Defacing Facebook: A Security Case Study, 2007. Presented at the *USENIX Security '07* Poster Session.

[21] Felt, A., and Evans, D. Privacy Protection for Social Networking Platforms. In *Proceedings of the Web 2.0 Security and Privacy Workshop (W2SP)* (2008).

[22] Glazman, D. Monetizing Firefox Extensions. In *Proceedings of the Mozilla Add-ons Workshop Berlin* (2009).

[23] Grier, C., King, S., and Wallach, D. How I Learned to Stop Worrying and Love Plugins. In *Proceedings of the 2009 Workshop on Web Security and Privacy* (2009).

[24] Kesan, J., Majuca, R., and Yurcik, W. Cyberinsurance as a Market-Based Solution to the Problem of Cybersecurity—A Case Study. In *Proceedings of Workshop on the Economics of Information Security (WEIS)* (2005).

[25] Kilpatrick, D. Privman: A library for partitioning applications. In *Proceedings of the USENIX Annual Technical Conference* (2003), pp. 273–284.

[26] Krishnamurthy, B., and Wills, C. E. On the Leakage of Personally Identifiable Information Via Online Social Networks. In *Proceedings of the Second ACM SIGCOMM Workshop on Social Network Systems (WOSN '09)* (2009).

[27] Landi, W. Undecidability of Static Analysis. *Letters on Programming Languages and Systems (LOPLAS) 1*, 4 (1992).

[28] Louw, M. T., Lim, J. S., and Venkatakrishnan, V. N. Enhancing web browser security against malware extensions. *Journal in Computer Virology 4*, 3 (2008), 179–185.

[29] MacDonald, E. The Council, the Computer and the Unfair Contract Terms Act 1977. *The Modern Law Review 58*, 4 (1995), 585–594.

[30] Murphy, B., and Levidow, B. Windows 2000 Dependability. Tech. Rep. MSR-TR-2000-56, Microsoft Research, 2000.

[31] Murray, D. G., and Hand, S. Privilege separation made easy. *the ACM SIGOPS European Workshop on System Security (EUROSEC)* (2008), 40–46.

[32] Necula, G. Proof-Carrying Code. *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (1997).

[33] Provos, N., Friedl, M., and Honeyman, P. Preventing Privilege Escalation. In *Proceedings of the 12th USENIX Security Symposium* (2003).

[34] Provos, N., McNamee, D., Mavrommatis, P., Wang, K., and Modadugu, N. The Ghost in the Browser: Analysis of Web-Based Malware. In *Proceedings of the First Conference on First Workshop on Hot Topics in Understanding Botnets (HotBots '07)* (2007).

[35] Saltzer, J. H., and Schroeder, M. D. The Protection of Information in Computer Systems. *Communications of the ACM 17*, 7 (1974), 1–30.

[36] Saroiu, S., Gribble, S., and Levy, H. Measurement and analysis of spyware in a university environment. In *Proceedings of the First USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2004).

[37] Schneier, B. Computer Security: It's the Economics, Stupid. In *Proceedings of Workshop on the Economics of Information Security (WEIS)* (2000).

[38] Seriot, N. iPhone Privacy. In *Black Hat DC* (2010).

[39] Shackman, M. Platform Security - a Technical Overview. Tech. Rep. v1.2, Symbian, 2006.

[40] Shavell, S. On Moral Hazard and Insurance. *The Quarterly Journal of Economics 93*, 4 (1979), 541–562.

[41] Watson, R. Exploiting Concurrency Vulnerabilities in System Call Wrappers. In *Proceedings of the First USENIX Workshop on Offensive Technologies (WOOT)* (2007).

[42] White, A. Caveat vendor? a review of the court of appeal decision in st albans city and district council v international computers limited. *The Journal of Information, Law and Technology (JILT) 3* (1997).

[43] XiaoFeng Wang, Z. L., and Choi, J. Y. SpyShield: Preserving Privacy from Spy Add-Ons. In *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID)* (2007), vol. 4637/2007 of *Lecture Notes in Computer Science*, pp. 296–316.

[44] Yee, B., Sehr, D., Dardyk, G., Chen, J. B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., and Fullagar, N. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proceedings of the 30th IEEE Symposium on Security and Privacy* (2009), pp. 79–93.

[45] Yee, K.-P. Aligning Security and Usability. *IEEE Security and Privacy Magazine 2*, 5 (2004), 48 – 55.

[46] Zittrain, J. *The Future of the Internet (And How to Stop It)*. Penguin, 2009.