Security Issues for Internet Appliances

Frank STAJANO[2,1]
[1] Laboratory for Communications Engineering
University of Cambridge
Cambridge, United Kingdom

Hiroshi ISOZAKI[2]
[2] Communication Platform Laboratory
Toshiba Corporate R&D Center
Kawasaki, Japan

Abstract

Internet-connected consumer appliances, under a dedicated and simplified user interface, often contain fully functional computers, sometimes even based on standard software platforms and operating systems. These appliances are therefore vulnerable to the same threats that plague desktop computers, including viruses and worms. Owing to their connectivity, uniform configuration and lack of professional administration, they also run a serious risk of being exploited for DDOS (distributed denial of service) attacks. This is where many previously infected appliances are woken up simultaneously by an evil master and ordered to bombard a designated victim from all sides.

In this paper we explore threats and defenses for this scenario. The issues we discuss include secure firmware upgrades, intrusion detection, remote administration and manufacturer liability.

1 Introduction

Ever since computers were first created half a century ago their size and cost have been steadily decreasing, bringing them from room-sized machines that only large corporations could afford to personal machines of which everybody could have one. As this trend continues, we are heading towards a scenario in which every user owns not one but dozens or even hundreds of computing devices. They won't all look like desktop computers, of course: part of the ongoing revolution consists of transforming computing capability into an inexpensive commodity that can be *embedded* into any device to enhance its functionality. So most of these devices will actually be computer- and communication-enhanced appliances as opposed to general purpose computers.

Weiser [16] coined the expression "ubiquitous computing" to denote a future scenario in which computing unobtrusively pervades our lives in the same way as writing, and is unobtrusively embedded in everyday objects in the same way as electric motors. (Can you identify all the electric motors embedded in the objects you own? They have *disappeared* from perception, and this is what will happen with computers.) An influential book by Norman [10] gives a commercial slant to these research ideas and makes the business case for the dedicated information appliance (specialized to do one thing well, and easily) as opposed to the general purpose computer (which can be used to perform many tasks but is often too difficult to use and look after).

Nakajima *et al.* [9] advocate the benefits of building Internet-connected appliances using standard software components such as Linux, CORBA and Java. In this paper we examine the security issues of such an arrangement.

2 Risks and benefits of standard software

The strategy of building an appliance on top of standard software such as Linux, CORBA and Java has definite advantages in terms of rapid product development: programmers exploit their existing know-how (system, libraries and development tools), code reuse is facilitated and documentation is plentiful. Perhaps one of the disadvantages is size: the existing system software, targeted to desktop systems with abundant memory and processing resources, may be too demanding for the appliance environment. This is one of the primary concerns addressed by the Embedded Linux community.

What about security? The problem of size affects security too; in fact it is probably one of the most dangerous aspects of using such standard software in an Internet appliance. Size in this context is meant not simply as memory footprint (which some providers of embedded Linux reduce with on-the-fly compression techniques) but rather as complexity, number of functions in the API, number of lines in the source—and, proportionally, number of outstanding bugs. The advantage of developing for a system supporting the full complement of familiar functionality and standard APIs must be paid for with the penalty of bringing in many seldom-used subsystems whose functionality will only be of use to the crackers who find exploitable bugs in them.

It would therefore be desirable to prepare and maintain minimalist versions of these standard software components (particularly the OS and associated utilities) with the explicit goal of reducing not just memory occupation but also complexity and functionality. The lucky circumstance that these goals are actually compatible should be an incentive.

One is reminded of the traditional computer security wisdom about the TCB (Trusted Computing Base) [2], the core component upon which the security of the whole system depends. The TCB ought to be as small as possible, first to honour the principle of least privilege, and second to be amenable to close security scrutiny if not formal verification.

The trouble with this approach is that the programmer who is given a supposedly secure core which only offers limited functionality is going to be forced to add in (or, worse, write from scratch) extra libraries and components until she can implement the functionality that is actually required of the appliance. Of course, if the crackers now find their holes at the application level rather than at the system level, the overall level of (in)security has not changed very much. It would be naive to imagine that making the system software tight and tiny could get rid of all security problems: it is certainly a helpful step, but it needs to be integrated in a holistic view of the production process, under the guidance of a well-defined security policy [3].

Another key observation on the security of using embedded Linux for Internet appliances is the one about open source. This is the well-known debate about the relative merits of "full disclosure" versus "security by obscurity". Proponents of the latter argue that, with open source, crackers have a much easier time discovering hidden vulnerabilities. If the code is kept proprietary, the cost of the attack goes up, because crackers have to "guess" holes and verify their presence by experiment, which is much more time-consuming than finding them by inspection of the source. Supporters of the former strategy, conversely, point out the importance of competent peer review, impossible in a closed source system, and argue that, within the open source framework, users will cooperate in finding bugs early. In his influential analysis of the open source development process, Raymond [12] famously observed that "with enough eyeballs, all bugs are shallow". Assuming axiomatically that any software will have bugs and security holes, the open source approach makes finding these bugs easier for everyone, whereas the closed source approach makes it hard for developers (because the user community cannot support them in the same way) and even harder for outsiders (because they can only observe the system as a black box).

It is true that the peer review process for Internet appliances is going to be radically different from the one for Linux or Apache. Users of Linux are mostly programmers who enjoy getting their hands dirty and take pleasure and pride in reading source code; but the demographics of the users of Internet appliances are going to be radically different, and it would be unrealistic to expect the average cyberfridge user to send in a bug report with a patch.

Still, the traditional cryptologic wisdom, going back to 1883 with Kerckhoffs [8], squarely supports what we now call open source: security should reside in the key and not in the method—because the method will ultimately be known by the adversary anyway. So, while keeping the method secret may delay an attack, relying on the method's secrecy to *prevent* attacks will only provide a false sense of security and is a course of action that is doomed to failure.

3 Malicious code

One of the major vulnerabilities for a computer system is the possibility that an attacker might run malicious code on it. Such malicious code could execute any conceivable action; therefore the host system is subject to a full spectrum of threats, covering all the categories in the traditional taxonomy: confidentiality, integrity and availability. The malicious program could disclose information found on the host system (like the Sircam email worm that made the rounds in July 2001 [6]), thereby violating confidentiality; it could surreptitiously alter data files, violating integrity; it could steal system resources so as to slow down the system, violating availability (many viruses do that anyway as a side effect of their reproductive activity); or delete files altogether (perhaps the most classical of virus payloads), affecting both integrity and availability.

The well-established taxonomy mentioned above focuses on *information* security; but the ill effects of malicious code can reach even beyond that. If the system is connected to actuators of any kind, the actions of the attacking code can have direct physical effects on the outside world. Without going to the dramatic extremes of what might happen to a factory plant or an airplane, an Internet appliance has the potential to spoil food, injure its user or set fire to the house.

A further threat from malicious code is that the Internet appliance might be used for Distributed Denial Of Service (DDOS). Attack software is installed on the appliance and left dormant, to be woken up at a later time by some evil "master" once a sufficient number of appliances have been infected. At that point all the infected appliances are ordered to attack a particular target simultaneously. Even if the individual attack consists of nothing more than sending messages to the victim, its replicated nature will saturate the victim's upstream connection. The attack is debilitating and there is very little that the victim can do other than disconnecting, which denies connectivity anyway. Gibson [7] provides interesting technical details about a DDOS attack to his web site; his report is interesting also because of the

insights it offers into the psychology and motivation of the perpetrators of the attack. One should not believe that attacks will only happen if they bring financial gain to their authors.

Internet appliances are going to be a particularly attractive target for attackers wishing to spread DDOS agents because the appliances will be permanently connected to the network and will usually not have a knowledgeable system administrator looking after them and noticing security breaches. This should be kept in mind before attempting a wide scale deployment.

4 Firmware upgrades

Since malicious code can be such a dangerous attack tool, it might appear attractive to ban the execution of foreign code altogether. But what should we then do if we discover a security hole—should all instances of the appliance be recalled to the manufacturer? The cost of this proposition may push us towards making the appliance field-upgradable, perhaps through its very own Internet connection. However, any mechanism through which new firmware can be uploaded in the appliance could potentially be exploited by malicious code too, in which case the appliance would be completely taken over.

An obvious fix to this problem, in the straightforward case in which the only acceptable firmware upgrades are those from the original manufacturer[1], is to adopt digital signatures. Each appliance holds in its ROM a copy of the public key of the manufacturer, which it uses to verify any downloaded firmware upgrades before installing them.

It would seem prudent to keep the signature verification code in ROM too, so that it can never be rewritten; this way, even if a small piece of malicious code somehow managed to penetrate the appliance in some other way, it would be prevented from bootstrapping itself into installing a full (counterfeit) firmware upgrade. However it is unclear whether, by itself, this additional countermeasure buys us a great deal of additional security: once the malicious code is in control, it could download the counterfeit upgrade even if it didn't have a good signature, since nothing actually forces the malicious code to even *call* the signature verification procedure that is safely locked down in the ROM. We must ensure not only that nobody can tamper with the signature verification software, but also that that software is actually run every time a new firmware installation is attempted.

To achieve this goal we hereby propose the **Cyclical Suicide** architecture model, in which the appliance periodically resets itself and jumps back in the tamperproof[2] ROM code.

There are two principal components to this model. One, an unstoppable countdown timer similar to the one we proposed in the *grenade timer* construction [13], regularly brings the flow of control back to a known-good portion of the code. The other is a low level address-based access control mechanism that ensures that the flash memory hosting the firmware can only be written from within the trusted ROM code.

The system comprises the following three kinds of memory.

ROM, containing a copy of the manufacturer's public key and the trusted code that can check signatures and install a firmware image into flash.

Flash, containing the currently installed and active firmware image—which includes operating system, middleware and application.

RAM, used as normal working memory for the system but also containing a reserved buffer in which new firmware images are stored prior to being installed into flash.

The ROM code, executed on reset, performs the following tasks in order.

1. It checks the RAM buffer for an incoming firmware image. If it finds one, it verifies its signature against the public key of the manufacturer and whether the manufacturing date is later than the date of the image currently installed in the flash memory.

2. If both checks pass, the incoming image is valid and fresh, and ought to be installed. So it is copied to the flash, overwriting the previous image. If not, this step is ignored.

3. The code then jumps into the flash, yielding control to the installed firmware image.

There is a countdown timer that unconditionally resets the processor when it reaches the end of its count. On reset, this timer is reloaded from a value stored in the ROM. There is no way for software to reload the timer at any other time (this makes this component more similar to our *grenade timer* than to the traditional *watchdog timer*), thereby lengthening the interval between forced resets, but there is provision for invoking a reset independently of the timer, thereby shortening that interval.

There is a hardware-enforced address protection mechanism such that it is not possible to write to the flash other than from the ROM. This means that no code executing from RAM can install itself (or anything else) permanently

[1] As opposed to, say, propellerhead-friendly appliances that can be reprogrammed by the user.

[2] "Tamperproof" here means "that cannot be modified from software". We are guarding from attacks staged by malicious incoming code, not from situations where the attacker has physical control over the appliance.

in the appliance, and it will not survive the next reset. The only way that new software can be permanently installed is by placing a firmware image with a valid signature in the appropriate place in the RAM buffer, and then invoking a reset.

Software will usually dislike being reset by the timer in the middle of its operation, and this is the principal drawback of this approach. We envisage that appliances will spend most of their time idle and have infrequent bursts of activity. So one strategy is for the appliance to check, after completing each burst of activity (i.e. at those points in time when it would be safe for it to be reset), how much time remains before the next reset; if this time were too short, the appliance could voluntarily reset itself so as to start afresh with a full-length period. But this, of course, only *reduces* the probability that the appliance will be reset in mid-activity, without eliminating it. So it will still be necessary to ensure that the software of the appliance is robust towards this sort of rough treatment.

Another disadvantage is that the appliance (or at least its computing subsystem—hopefully it will still be possible to, say, open the door of the cyberfridge) is going to be unresponsive during the reset. With PCs still taking several minutes to boot despite processor speeds in the GHz region, the suggestion of frequent reboots does not appear as very attractive. Clearly the duration of a reset had better be small compared to the uptime allowed by the timer. On the other hand we wish the uptime to be small since it is the window of vulnerability during which any intruding malicious code can remain in control of the appliance. The trade-off between these two goals will have to be managed carefully. We also hope that a system streamlined for embedding, and with no moving parts in its mass storage, will be able to boot in much less time than a standard desktop PC.

This same mechanism can be used to install incremental patches as opposed to full firmware images. A patch, carrying a hash of the image it is supposed to modify, is signed in the same way as an image. In step 1 above, the ROM code also checks that the hash matches the currently installed image. In step 2, instead of a straightforward copy, the existing image is patched, using whatever algorithm is appropriate to the format of the patch.

We recommend that the manufacturer use a different signing key for each model of appliance. To use the same key for all products would yield no benefit, while increasing the potential damage (and therefore the incentive to steal that key) if that key were ever compromised.

The grenade timer [13], which inspired the Cyclical Suicide model, was developed for a situation in which the CPU did not have a protected mode and was therefore incapable of restricting the behaviour of an external program. Some might feel that in the present case additional hardware protection is overkill because the processor already has a protected mode (or it couldn't run Linux). We must however notice that the threat is not simply from mobile code that is officially recognized as such. The grenade timer was designed to ensure that guest mobile code could not step out of its allowed boundaries, but we must be prepared to fight malicious code that we have not recognized as coming from outside.

For example, one of the most widespread attack routes against networked machines is the *buffer overflow*, in which an input routine which does not check the size of its input[3] is fed an oversized piece of data, the tail of which overwrites the stack and therefore gets executed as code, at the same privilege level as the program of which the input routine was part.

There are more variants to this basic scheme: we won't go into details, but we refer the interested reader to Cowan *et al.* [5], who offer an extensive survey of buffer overflow vulnerabilities, as well as some software countermeasures and an analysis of their associated performance penalty. As they observe,

> All buffer overflow vulnerabilities result from the lack of type safety in C. If only type-safe operations can be performed on a given variable, then it is not possible to use creative input applied to variable foo to make arbitrary changes to the variable bar. If new, security-sensitive code is to be written, it is recommended that the code be written in a type-safe language such as Java or ML. [...] However, it is also the case that the Java Virtual Machine (JVM) is a C program, and one of the ways to attack a JVM is to apply buffer overflow attacks to the JVM itself [...].

The Cyclical Suicide model, because it works at a lower level than the OS, blocks write attempts to the flash independently of the privilege level of the piece of code that requests them, granting permission only to the immutable piece of code contained in the ROM. It also limits the lifetime of any RAM-based program, which prevents malicious code from taking over the appliance permanently.

5 Intrusion detection

Defense against security threats can take place before or after the fact—Amoroso [1] speaks of *safeguards* and *countermeasures* respectively. While safeguards are preferable when possible, insofar as they actually prevent an attack, it is often impossible to safeguard a priori against every possible attack. For the remaining attacks, then, a posteriori countermeasures are appropriate, and to apply them one needs to be aware that an attack is taking place.

[3] This includes most common uses of the standard C library functions `gets()` and `scanf()`.

Tools for the detection of attacks include Intrusion Detection Systems (IDS): these analyze usage patterns, detect security violations and notify the users or administrators when appropriate.

As explained by Northcutt [11], we can identify two main approaches to detect intrusion. One is "Misuse Detection": we define a precise sequence of events or a specific pattern that has been observed in malicious activities and we look for it. The other is "Anomaly Detection": we first create profiles that describe normal system usage and then we detect any significant deviation from the profiles.

"Misuse Detection" is the strategy that is commonly adopted by antivirus programs for PCs: the vendor continually analyzes any new viruses that are discovered and, for each one of them, prepares an identifying "signature" that allows the scanning engine to detect that specific virus. The user is then required regularly to obtain the most recent set of signatures from the antivirus vendor.

In the context of Internet appliances, though, we will probably not be able to prepare signatures before the system is attacked by viruses because, in a standardized environment and with machines that have a permanent net connection, viruses have the potential to spread much more quickly than has traditionally happened in the PC environment. On the other hand, PC malware is getting quicker too: in late summer 2001 there were speculations about "Warhol worms" [15] (infecting the Internet in 15 minutes) and even "Flash worms" [14] (doing the same in 30 seconds), which accelerate their propagation by predetermining the list of machines to infect. The technique is clearly applicable to appliances.

Most of the IDSs in the market today adopt a (deterministic) misuse detection algorithm because this causes fewer false positives[4] than an anomaly detection algorithm based on heuristics. But this strategy suffers from the same fatal defect as virus scanners. Misuse detection compares current system usage with a database of signatures, so systems must always update their signatures to catch new attacks. In contrast anomaly detection algorithms are useful and powerful against unknown attacks.

Although many researchers have tried to apply anomaly detection heuristics to intrusion detection, most of these attempts are not yet ready for deployment. The accuracy of anomaly detection depends on the detection algorithm and on the appropriate selection of the system features to be profiled, for example CPU usage or network activity. Each system runs in a different environment and has different users. The IDS manager is faced with the difficult task of selecting features that describe the behaviour of the system appropriately and exactly.

[4] A false positive is a "boy crying wolf" false alarm: the IDS wrongly claims that there is a problem, but no attack is in progress. A false negative, instead, is a failure to detect an attack.

Although adopting an anomaly detection algorithm for an Intrusion Detection System is hard in general, we suggest that it will be a good choice for Internet appliances, because their behavior is more limited and predictable than that of a general purpose computer. For example, on a general purpose computer, users perform many different tasks such as editing files, writing e-mails, browsing web sites, etc. This makes it difficult to create profiles for normal system usage. On the contrary, for Internet appliances, configuration, hardware components and usage are fixed or at least very restrained, and users do not usually have access to facilities to modify the system. It will therefore be easier to define normal usage patterns and consequently detect unexpected actions such as those that might be caused by attacks.

6 Remote administration

Intrusion detection systems need to be managed: a competent party needs to pay attention to the alerts and take appropriate action. This might seem inappropriate for Internet appliances, which are going to be operated by non-technical users. However we may address this problem with a remote administration service offered by the manufacturer or by an independent service company. Outsourcing the administration burden is one way to reduce system complexity for the end user.

By centralizing the alerts collected from the various appliances, manufacturers will be able to know what attacks are going on, estimate the extent of damage and prepare adequate patches for security flaws. Remote administration will allow timely distribution of such patches in order to upgrade the firmware of the deployed units.

There is one subtle problem with remote administration: respecting user privacy. This is not trivial, and it will be easy to misuse remotely-controlled appliances for surveillance purposes unless specific anonymization and obfuscation safeguards are built into them by design. For example, even the usage logs of a coffee pot or refrigerator could be analyzed to infer whether the owner is at home or on holiday, or whether the household has any extra visitors this week. This invasion of privacy needs to be addressed at the design level by appropriately limiting the information to which the remote administrator has access. Inference control is a well-known hard problem in database security.

A related and equally difficult problem is that of assurance: even if the system has been designed to restrict the amount of information available to the remote administrator, how can the user believe that this is the case? Open source and hostile peer review, discussed in section 2, are part of the solution; but, even so, it will be difficult for the user to verify that the code inside the appliance is the same as that described in the published source.

7 Liability

Perhaps some of the most important questions about the security of Internet appliances are not technical but legal. We are used to licensing agreements in which software manufacturers disclaim all liability for any damages caused by their software: the makers of insecure email clients that are regularly exploited to spread viruses have not, as yet, been sued by users. But what will be the rule for Internet appliances? After all, the makers of conventional appliances *are* (perhaps more fairly) occasionally brought to court when their products put their users in danger or bring them losses.

As we hinted at in section 3, appliances have physical actuators and therefore have the potential to cause damage not only in cyberspace but also in the physical world: the heating elements of a toaster, the spinning blades of a liquidizer, the magnetron of a microwave oven could all behave dangerously to the user. More subtly, if the motor of a fridge were stopped at irregular intervals, the spoilt contents might cause food poisoning.

As a baseline technical defense it seems prudent to avoid the "soft" approach in which all the raw functions of the appliance can be controlled from software. There should instead be hardware interlocks that make it impossible, say, for microwaves to be emitted unless the door of the oven is closed. However this will never be a total solution against all attacks. Think of the fridge example above: is it reasonable to exclude software control of even the power switch? Or even better: think of a hifi system that a virus could force to play a siren sound at full volume in the middle of the night. Surely software should be allowed to start and stop playback, and to turn the volume up and down, so it's hard to imagine a protective interlock that would make sense; and even if the system were prevented from handling digitally downloaded sounds such as the siren (which in itself seems pretty unlikely for an Internet-connected hifi), playing *anything* at full 150 watt volume in the middle of the night would be enough to give someone a heart attack, even if it's just their favourite classical music CD.

Going back to traditional, non-Internet appliances, the fridge maker will not get sued if food gets spoilt because of a power outage; however it probably will if food gets spoilt because of a design flaw that makes the thermostat unstable. What counts is not just the effect, but the responsibility.

It is likely that some manufacturers will attempt to deny liability by shifting the blame on the customer for having used the appliance improperly. But they are unlikely to win the sympathy of the court unless they can demonstrate that they took all reasonable precautions to avoid the threat. An interesting paper by Anderson [4] provocatively highlights how security is added to banking systems not simply to reduce risk but to transfer liability to some other party—often the customer ("our PINs are generated and printed inside secure hardware, so any disputed withdrawals must be the customer's fault"). Fortunately, as a defense expert, he also reports success in protecting defrauded customers during legal disputes by asking that the bank disclose its (often less than perfect) security procedures. He summarizes the lesson as follows:

> **Principle 1:** Security systems which are to provide evidence must be designed and certified on the assumption that they will be examined in detail by a hostile expert.

Similar considerations are likely to hold for appliances. While manufacturers should certainly reject false claims by fraudsters who disconnected their own fridge and claimed the food was spoilt by a malfunction or a virus, they should not expect to be able to deny the possibility of any security problems with their Internet appliances simply because their firmware "includes encryption".

Given that it will be impossible to prevent all attacks, part of the design effort should go towards limiting the maximum damage that the system can cause if attacked. To quote Anderson [4] again,

> **Principle 9:** A trusted component or system is one which you can insure.

8 Conclusions

The security problems faced by Internet appliances are many and varied. If an Internet appliance is built like a PC, even using the same commodity software components, we should expect the standard PC security problems to resurface. We support full disclosure and the open source approach as methodologies that stand a greater chance to discover and correct security holes in a timely fashion.

In this paper we have focused primarily on the problems caused by malicious mobile code such as viruses and worms. We have proposed the *Cyclical Suicide* architecture model to prevent such malicious software from infecting the machine permanently during a firmware upgrade. We have also shown that remotely managed intrusion detection systems may constitute an effective solution for appliances, much more so than for PCs.

Finally, addressing security problems is an activity that goes beyond the domain of technology: at the design stage, manufacturers ought to attempt to limit the maximum damage that the appliance can perform if it goes wrong. At the same time, the debate is open on the issue of liability, which software vendors have so far skillfully dodged: if an appliance is attacked and cracked, perhaps all too easily, how much of the blame should be borne by the manufacturer?

References

[1] E. Amoroso. *Fundamentals of Computer Security Technology*. Prentice Hall, 1994.

[2] R. Anderson. *Security Engineering—A Guide To Building Dependable Distributed Systems*. John Wiley & Sons, 2001.

[3] R. Anderson, F. Stajano, and J.-H. Lee. Security policies. volume 55 of *Advances in Computers*. Academic Press, 2001.

[4] R. J. Anderson. Liability and computer security: Nine principles. In D. Gollmann, editor, *Proc. 3rd European Symposium on Research in Computer Security – ESORICS '94*, volume 875 of *Lecture Notes in Computer Science*, pages 231–245. Springer-Verlag, Nov. 1994. http://www.cl.cam.ac.uk/ftp/users/rja14/liability.pdf.

[5] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *Proceedings of the DARPA Information Survivability Conference & Exposition Volume II of II*. IEEE, 1998. http://dlib.computer.org/conferen/discex/0490/pdf/04901119.pdf. Access to the URL requires subscription.

[6] P. Ferrie and P. Szor. W32.sircam.worm@mm, 2001. http://www.symantec.com/avcenter/venc/data/pf/w32.sircam.worm@mm.html.

[7] S. Gibson. The strange tale of the denial of service attacks against grc.com, 2 June 2001. http://media.grc.com:8080/files/grcdos.pdf.

[8] A. Kerckhoffs. La cryptographie militaire *(Military Cryptography)*. *Journal des sciences militaires*, IX:5–38, Jan. 1883. http://www.cl.cam.ac.uk/~fapp2/kerckhoffs/. In French. The second part of the article appears in the Feb 1883 issue, pp. 161–191.

[9] T. Nakajima, H. Ishikawa, E. Tokunaga, and F. Stajano. Technology challenges for building internet-scale ubiquitous computing, 2001. To appear.

[10] D. A. Norman. *The Invisible Computer: Why Good Products Can Fail, the Personal Computer Is So Complex, and Information Appliances Are the Solution*. MIT Press, 1998.

[11] S. Northcutt. *Network Intrusion Detection—An Analyst's Handbook*. New Riders, Indianapolis, IN, USA, 1999.

[12] E. S. Raymond. *The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly & Associates, Inc., 1999.

[13] F. Stajano and R. Anderson. The grenade timer: Fortifying the watchdog timer against malicious mobile code. In *Proceedings of the* 7^{th} *International Workshop on Mobile Multimedia Communications*, Waseda, Tokyo, Japan, Oct. 2000. http://www-lce.eng.cam.ac.uk/~fms27/papers/grenade.pdf. Also available as AT&T Laboratories Cambridge Technical Report 2000.8.

[14] S. Staniford, G. Grim, and R. Jonkman. Flash worms: Thirty seconds to infect the internet, 16 Aug. 2001. http://www.silicondefense.com/flash/.

[15] N. C. Weaver. Warhol worms: The potential for very fast internet plagues, 15 Aug. 2001. http://www.cs.berkeley.edu/~nweaver/warhol.html.

[16] M. Weiser. The computer for the twenty-first century. *Scientific American*, 265(3):94–104, Sept. 1991. http://www.ubiq.com/hypertext/weiser/SciAmDraft3.html.