Remote Booting in a Hostile World

To Whom am I Speaking?

Mark Lomas
University of Cambridge Computer Laboratory

Bruce Christianson
University of Hertfordshire, Hatfield

Today's networked computer systems are very vulnerable to attack. The collision-rich hash function described here permits a secure boot across a public network with no security features.

In the spring of 1989, students at the University of Cambridge successfully penetrated the Computer Laboratory system. The attack on computers used as public area terminals was intricate and involved physically dismantling and replacing components with new firmware that recorded user passwords for later replay.[1] The laboratory responded by modifying the anti-theft devices to ensure that future hardware tampering would be evident to a careful user.

Today's networked computer systems are even more vulnerable to attack: Terminal software, like that used by the X Window System, is frequently passed across a network, and a trojan horse can easily be inserted while it is in transit. Many other software products, including operating systems, load parts of themselves from a server across a network. Although users may be confident that their workstation is physically secure, some part of the network to which they are attached almost certainly is not secure.

Most proposals that recommend cryptographic means to protect remotely loaded software also eliminate the advantages of remote loading—for example, ease of reconfiguration, upgrade distribution, and maintenance. For this reason, they have largely been abandoned before finding their way into commercial products.

This article shows that, contrary to intuition, it is no more difficult to protect a workstation that loads its software across an insecure network than to protect a stand-alone workstation. Flexibility is not sacrificed with our solution, nor are users required to trust the integrity of any part of the system that they cannot physically see or control.

THE PROBLEM

Figure 1 depicts the problem. We have a large number of workstations scattered among insecure sites such as homes, offices, and perhaps even public areas. The workstations will be used by different, and possibly mutually suspicious, individuals. Each workstation runs an operating system kernel, which may include access to a security policy service. Each kernel is trusted by particular users for certain purposes—for example, share trading, market forecasting, or software development. Users want to be certain that the correct kernel is running on any workstation that they use. But there need be no global kernel trusted by all users for all purposes, or by all users for any single purpose. Further, a single user may require a number of different kernels trusted for different purposes.

We assume that the kernel cannot be placed in ROM because of size constraints, or because the user must be able to swap kernels without replacing the ROM. Therefore, the workstation must download the kernel from elsewhere—for instance, from a boot service accessed across a network. How can we ensure the integrity of the downloaded code?

0018-9162/95/$4.00 © 1995 IEEE

The most obvious approach to the problem involves securing the entire network, including all the boot servers. This would, however, require the network addresses of all possible boot servers to be hard-wired immutably into ROM when the hardware is assembled. We will show that this approach is unnecessary, as well as impractical.

An alternative to securing the network is an end-to-end approach[2] that ensures that the code, however loaded, is correct prior to passing control to it. This approach has the added benefit that the user does not need to trust the boot code (including network drivers) responsible for downloading the kernel, which can therefore be placed in RAM and changed (maintained) at will. This approach also renders it unnecessary for us to trust the boot server, as we can detect when a server misbehaves.

DEFINITIONS AND ASSUMPTIONS

We are considering the problem of securely booting a relatively stateless workstation in a potentially hostile environment. By secure booting, we mean that the user initiating the boot requires a high degree of justified confidence that the code loaded into the workstation as a result of the boot is code that the initiating user trusts to act correctly. In other words, the user is prepared to bear the risk of the right code acting wrongly, but not to bear the risk of the wrong code being loaded. By relatively stateless, we mean that the workstation, while unattended, cannot preserve with any degree of reliability the integrity of mutable data (that is, data that can potentially be changed without replacing the hardware).

By a potentially hostile environment we mean that the network to which the workstation is attached, and all the services accessed across the network, are subject to interference by chance or by deliberate attack. In addition, we do not rely on the integrity of any boot code loaded locally into the workstation, such as network device drivers.

Our definitions imply that we cannot rely on the workstation to preserve the integrity of a secret, such as a password or cryptographic key, since either may be compromised and therefore need to be changed. We also assume that the initiating user, being human, cannot reliably recognize or remember a well-chosen key (one with high entropy), whether completely private, shared-secret, or public. On the positive side, we do assume that the user can remember and not reveal a poorly chosen password (one with low entropy).

We assume that the workstation is tamper-evident: The keyboard, CPU, RAM, and ROM hardware (and interconnects) are physically sealed at the time of manufacture so that a careful user would notice subsequent hardware alteration. Hence, the user can trust this workstation hardware to function correctly—or leastways, the way it did when the manufacturer tested it.

Next, we assume that the initial ROM contents cannot be changed by the user or anyone else under any circumstances. Thus, the workstation can maintain the integrity of a small amount of immutable code (including a hash function and a keyboard driver).

If there were a legitimate way to change the ROM contents, this would potentially allow an attack based on misuse of the change method. Our assumption that the ROM cannot be legitimately changed provides automatic protection against any such attack. Our assumption also makes it easier for the manufacturer to build tamper-evident hardware and provides convenient maintenance and configuration management.

We assume that the user can force the workstation into a known initial state at will, that is, that there is a conceptual red button the user can press to set the program counter to a fixed ROM address. This must also disable any other interrupts or untrusted hardware components such as DMA communications and secondary storage devices.

Finally, we assume that the keyboard is secure, by which we mean (roughly speaking) that the keyboard doesn't reveal what is typed on it except to the workstation CPU. Thus, the workstation can preserve the secrecy of a password entered from the keyboard by forgetting or erasing the password prior to transferring control to any mutable code in RAM, such as boot code.

Now we show how to perform a secure boot under these conditions.

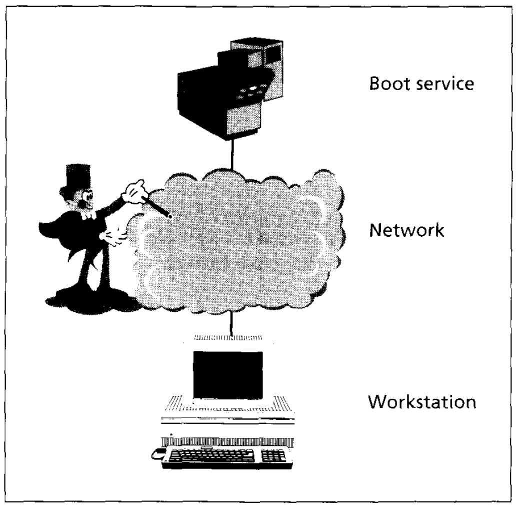

Figure 1. Remote-booting scenario.

SOLUTION STRATEGY

The simplest way to implement the end-to-end approach is to precalculate a checksum (hash) of the correct kernel code and to check that the loaded code has the correct checksum. But how can we be sure at boot time what the correct checksum is?

Collision-free hashing

If the checksum is long enough and sufficiently collision free (meaning that it is hard to find different data sets with the same checksum) to provide a strong guarantee of integrity, then by our assumptions neither the user nor the workstation can be relied on to remember the correct value of the checksum, since the checksum has high entropy and changes periodically when the kernel is updated by some party whom the user regards as competent to do so. Moreover, it is dangerous to store and download the checksum with the kernel code, since the hash function is publicly available. An attacker could therefore modify the kernel and recalculate the appropriate hash value. As a safeguard, we could require the party responsible for maintaining a particular kernel to sign the hash value using a public-key cryptographic system such as RSA. The problem is that neither the user nor the workstation can reliably remember the appropriate public key, since this key has high entropy and will change abruptly if the keyholder believes it has been compromised.

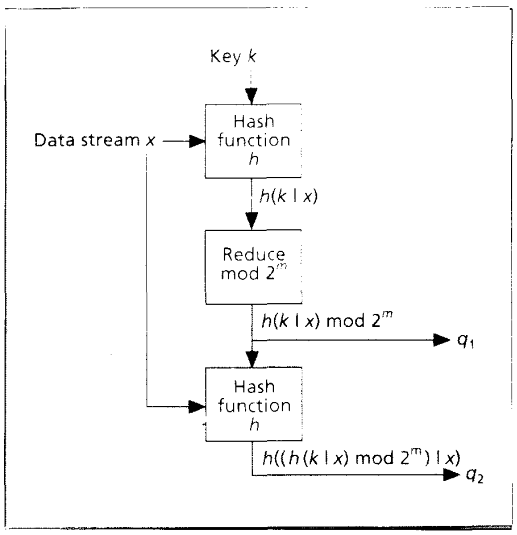

Figure 2. A collision-rich hash function.

Passwords

Some security could be provided by using a poorly chosen (low entropy) unshared secret, commonly called a password, which we assume is known only to a single user and used only for this purpose. The user can, with this password, maintain the integrity of data to be downloaded (for example, a checksum or a public key) by performing the following steps prior to the first download.

First the user must obtain, on a secure machine, an authentic copy of the data to be protected against modification. Users can do this by any mechanism which they (or their security policy) are willing to trust. Next, the password is hashed together with the data to produce a checksum. Finally, this checksum is appended to the data, and both are placed in a public area.

When the data is downloaded, the checksum is recalculated, using the password entered from the keyboard, and compared with the checksum appended to the data. If the two checksums match, and provided that the hash function is collision free, there is a high probability that either the data is unmodified or the password is compromised.

Unfortunately, the second of these possibilities is quite likely if we use a conventional collision-free hash function. A determined opponent can make an off-line guessing attack by downloading the data, then repeatedly guessing the password and calculating the checksum. Since the hash function is collision free, a match indicates to the opponent a high probability that the password has been correctly guessed. Since the password is poorly chosen, an opponent most likely has the computational resources to perform an exhaustive and successful search. The opponent is then able to modify the data in a way that will not readily be detected by the protocol in operation at the workstation.

However, we can defeat this attack by using a different type of hash function, deliberately chosen to provide numerous collisions while still providing a strong guarantee of integrity for the data. The idea is to ensure that an exhaustive off-line search by the opponent will produce not one, but many candidate passwords. Any one of these will produce the correct checksum for the correct data, but only one will produce the checksum expected by the user for the bogus data as modified by the opponent.

As we see later, the effect of this strategy is to ensure that any guessing attack by the opponent is effectively forced on line, in the sense that the attack now requires the user's interactive participation. This allows the user to offer a defense, which an off-line attack does not. Similar approaches are applied to related problems in Gong et al.[3] and in Anderson and Lomas.[4]

Collision-rich hashing

Before discussing a secure boot protocol, we offer a way to construct an appropriate collision-rich hash function from a conventional collision-free hash function (Figure 2).

Suppose that h is a collision-free hash function. Then, the hash function q defined by $q(k, x) = h((h(k \mid x) \bmod 2^m) \mid x)$, where $\mid$ denotes concatenation, will have the properties that we require, provided m is suitably chosen and h has suitable mixing properties. (For further details, see Berson, Gong, and Lomas.[5]) It is the reduction modulo 2^m that generates the deliberate collisions. The outer reapplication of h will restore the strong guarantee of integrity. We consider the choice of an appropriate value for m at the end of this section.

Now suppose that x is the data whose integrity the user wishes to protect, k is the user's password, and the checksum $q(k, x)$ is appended to x. The attacker wishes to modify the data in some way and construct a checksum for the modified data that will pass the user's validation check. But now there is not enough information to allow the attacker to determine the password uniquely. The attacker must guess the password (which is at least a better bet than guessing the checksum directly). If the attacker guesses wrongly, the user will become aware of the attack. Of course, the user may wrongly attribute the checksum mismatch to a network error or to a dirty sector on the boot server disk. But if the data x is followed by both $q_1 = (h(k \mid x) \bmod 2^m)$ and $q_2 = h(q_1 \mid x) = q(k, x)$, the user can almost certainly tell the difference between chance and deliberate attack: If $q_1 \neq (h(k \mid x) \bmod 2^m)$ but $q_2 = h(q_1 \mid x)$, then an attacker is almost certainly at work.

To prevent the attacker from obtaining the user's password by repeated guessing, the user should change his or her password immediately upon detecting an attack of this kind. Consequently, as with most defenses against an online penetration attack, there is now the risk of a denial-of-service attack. In this, the attacker deliberately corrupts the value of q_1 and recalculates q_2 to prevent the user from using any workstations, possibly in the hope that the user will eventually respond by ignoring the integrity failure and proceeding regardless. And this is precisely what the user should give the *appearance* of doing to ensure that the attacker gains no information about the correctness of the guessed password. The user is now aware that an attack is being made, and thus knows not to continue to rely on that password to protect the data's integrity.

As well as changing the password upon detecting an attack, the user must also change it whenever the protected data changes; otherwise, the attacker will have two independent pieces of information about the password. This will reduce the number of possibilities for k revealed by exhaustive search to a dangerously small value—typically, one. Because of this, the user's burden is lessened if the data protected directly by the password changes infrequently. Rather than use the password directly to protect the integrity of mutable data, it is better to hash the data with a collision-free hash function, and sign the hash with a high-entropy private key. The password then protects the integrity of the corresponding public key (and of any cryptographic code necessary to verify the signature). Together, the public key and cryptographic code can verify the mutable data. To ensure that the mutable data is fresh, a date stamp should be appended prior to hashing.

Password choices

A number of available tools will generate, from a uniform distribution, a syllable sequence that looks and sounds like an English word but isn't. For example, the Concept Laboratories' password generator,[6] if asked to generate a 12-letter password, will give one with an effective entropy of slightly over 28 bits. (A password has an effective entropy of m bits if the password was equally likely to have had any one of 2^m different values. Allowing the user to choose the password results in a lower entropy, since some choices are more likely than others. In either case, the effective entropy is considerably less than the bit-length of the password.)

Assuming that the user password k is generated in a similar fashion and has an effective entropy of $2m$ bits, then an exhaustive search for k by an attacker will (by our assumptions on q) reveal on the order of 2^m plausible passwords—values in the domain for k that satisfy $q(k, x) = q_2$. Consequently, the attacker has only a one in 2^m chance of correctly guessing the value of k employed by the user. Alternatively, the attacker could try to guess directly the correct value of $h(k \mid x') \bmod 2^m$ for the modified data x' and so deduce the values of q_1 and q_2 that would be accepted by the user as a guarantee of integrity for the bogus x'. However this attack also has only a one in 2^m chance of success.

A SECURE BOOT PROTOCOL

In an example of a secure boot protocol for a relatively stateless workstation, a user approaches a workstation and executes some untrusted local boot code. In response to user input, the local boot code initiates the secure boot protocol and downloads the desired kernel. The untrusted local boot code may load device drivers and various other bits of software from untrusted sources into the workstation before accessing the remote boot service via the network. After execution, the local boot code may or may not correctly load the following into the workstation:

1. the certification code (described below), including code to perform public key cryptography;
2. the public key of the authority responsible for maintaining the kernel;
3. the two hash values q_1 and q_2 as defined previously, applied to the concatenation of the data in the certification code (1) and public key (2);
4. the code for the secure kernel;
5. a certificate (Figure 3) for the kernel, consisting of (*i*) an identifier for the kernel, (*ii*) the value of a collision-free hash function h applied to the kernel code (4), and (*iii*) a date stamp (for freshness), signed under the private key corresponding to the public key (2).

In keeping with our assumptions, the user now presses the conceptual red button to pass control to the immutable ROM code and inputs the password. The ROM code recomputes the hash values in (3) and then forgets (erases) the user's password. If the computed values match, the ROM code then passes control to the certification code (1) in RAM, which is now known to be acceptable to the user. The certification code first uses the public key (2), which is also now known to be acceptable to the user, to check the validity of the certificate (5). If this check succeeds, the certification code next computes the hash h of the kernel code (4), and checks this against the value (*ii*) in the certificate (5). If the certificate value agrees with the calculated value, then the certification code will interact with the user to check whether the correct kernel (*i*) has been loaded, and whether the date stamp (*iii*) is acceptable. If all is well, then the certification code will pass control to the kernel.

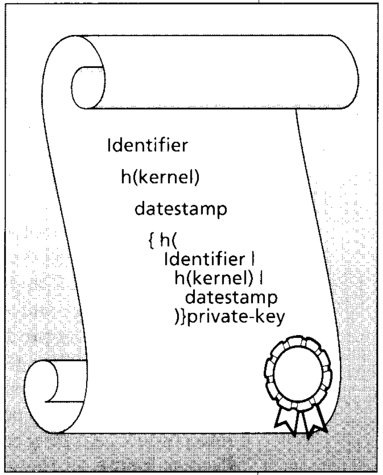

Figure 3. A kernel certificate.

Of course, the public key could be the user's own public key, which would allow the user complete control over the certificate. Any system code or device drivers loaded during the preliminary local boot and required to have integrity following the secure kernel boot need not be reloaded, but can simply be included in the kernel checksum in field (*ii*) of the certificate (5). A single password can also allow the use of variant public key cryptographic systems and key sizes if the h-hash of more than one piece of certification code is included in (1). Similarly, even if a user requires the flexibility of using kernels (or parts of kernels) maintained by many different authorities (with different public keys), still only one user password is required, since more than one public key can be included in (2).

A user can even add new public keys dynamically without changing the password (and without revealing two independent checksums calculated with the same password) by placing the user's own public key in (2), and then appending to each kernel certificate in (5) a proxy that contains:

- the kernel identifier,
- the public key of the appropriate authority for that kernel,
- a date stamp, and
- the signature—under the user's private key corresponding to the public key in (2)—of the concatenation of the data in the first three fields of the proxy.

This use of self-authenticating proxies is further developed in Low and Christianson.[7,8] Heterogeneity of hardware among workstations can also be accommodated in this way.

WITH OUR PROTOCOL, A USER CAN APPROACH A WORKSTATION previously used by a rival, perform a local boot from a floppy lying beside the workstation, and then download a system kernel and some RSA code from bulletin boards respectively maintained by a hackers' club and an intelligence agency, across a public access network with no security features. The user will still have highly justified confidence that the workstation is in the same state it would have been if the user had correctly entered the kernel manually. Although some users will doubtless continue to prefer the second option, it is pleasant to have the choice.

The ingenuity and motivation of those willing to attack sites with commercially valuable information should not be underestimated. The class of 1989 now has considerably more training and experience than it did then. Protocols such as the one described in this article change the situation somewhat. The removal of threats to system integrity from routine maintenance and from deliberate attack becomes simply a performance issue rather than a prerequisite for ensuring correct behaviour of the system. Whereas before an integrity failure could have catastrophic results, now it simply means that you can't boot up for a short time.

Our approach makes essential use of a hash function deliberately chosen to be rich in collisions. This contrasts with the prevailing practice of constructing hash functions that are as free as possible from collisions. We also make a complete separation of secrecy (read protection) and integrity (write protection.) We do not require of the workstation that tamper-proof data be kept secret, nor that secret data be protected from modification by untrusted code. All workstations with compatible hardware can therefore be treated as interchangeable from the hardware management point of view. Users are not required to trust their system managers, or any other part of the system that they cannot physically see or control. Computer manufacturers may also be pleased to learn that secure booting does not require serial numbers or cryptographic keys to be embedded in their machines, either during or after manufacture. ■

Acknowledgment

Part of this work was carried out while Mark Lomas was a visiting research fellow at the University of Hertfordshire.

References

1. D.D. Harriman, "Password Fishing on Public Terminals," *Computer Fraud and Security Bulletin*, Elsevier Science Publishers, New York, Jan. 1990, pp. 12–14.
2. J.H. Saltzer, D.P. Reed, and D. Clark, "End-to-End Arguments in System Design," *ACM Trans. Computer Systems*, Vol. 2, No. 4, Nov. 1984, pp. 277-288.
3. L. Gong et al., "Protecting Poorly Chosen Secrets from Guessing Attacks," *IEEE J. Selected Areas in Comm.*, Vol. 11, No. 5, June 1993, pp. 648-656.
4. R.J. Anderson and T.M.A. Lomas, "Fortifying Key Negotiation Schemes with Poorly Chosen Passwords," *IEE Electronics Letters*, Vol. 30, No. 13, June 1994, pp. 1,040-1,041.
5. T. Berson, L. Gong, and T.M.A. Lomas, "Secure, Keyed, and Collisionful Hash Functions," in Tech. Report SRI-CSL-94-08, SRI International, Stanford, Calif., May 1994.
6 J. Gordon, *Password Generation Software*, Concept Laboratories, Lynfield House, Datchworth Green, Hertfordshire, England, 1993.
7. M.R. Low and B. Christianson, "A Technique for Authentication, Access Control, and Resource Management in Open Distributed Systems," *IEE Electronics Letters*, Vol. 30, No. 2, Jan. 1994, pp. 124-125.
8. M.R. Low and B. Christianson, "Self-Authenticating Proxies," *Computer J.*, Vol. 37, No. 5, Oct. 1994, pp. 422-428.

Mark Lomas *is a research fellow at the University of Cambridge Computer Laboratory. His research interests include cryptography and computer security, which he also teaches as part of the Cambridge computer science tripos. He advises a variety of clients on computer and network security, with a special interest in financial systems. He obtained his BSc in computer science at the Hatfield Polytechnic (now the University of Hertfordshire, Hatfield) in 1984 and received his PhD in computer science from the University of Cambridge in 1992.*

Bruce Christianson *is principal lecturer in the School of Information Sciences at the University of Hertfordshire, Hatfield, United Kingdom. Prior to taking up his present post, he was a consultant with the communications business unit of Data Connection Ltd., specializing in electronic messaging and gateway design. He has been designated University Research Leader in the area of concurrent and distributed systems, and is also a member of the Numerical Optimisation Centre at Hatfield. He was awarded the BSc and MSc degrees in mathematics by the Victoria University of Wellington, New Zealand, in 1978 and 1980, and the DPhil degree in mathematics by the University of Oxford in 1984.*

Readers can contact Bruce Christianson at the Computer Science Division, University of Hertfordshire, Hatfield, AL10 9AB, UK; e-mail, B.Christianson@herts.ac.uk.