DIVISION OF COMPUTER SCIENCE

To Whom am I Speaking?
Remote Booting in a Hostile World

Mark Lomas
Bruce Christianson

Technical Report No.178

January 1994

To Whom am I Speaking? Remote Booting in a Hostile World.

Mark Lomas
(tmal@cl.cam.ac.uk)

Computer Laboratory, University of Cambridge, England, Europe

Bruce Christianson
(B.Christianson@herts.ac.uk)

School of Information Sciences, Hatfield Campus, University of Hertfordshire, England, Europe

January 1994

Abstract. We consider the problem of booting a workstation across a network. We allow "maintenance" (that is, change without notice by untrusted parties such as adversaries and system managers) to be freely performed upon the network, the workstation, and the remote boot service itself. We assume that humans are unable to recognise long sequences of independent bits such as cryptographic keys or checksums reliably, but can remember passwords which have been sufficiently poorly chosen to succumb to guessing attacks. We also assume that a part of the workstation hardware (including a small amount of ROM) can be physically protected from modification, but that the workstation cannot protect the integrity of any mutable data, including cryptographic keys (which must change if a secret is compromised.)

Nevertheless, we are able to provide strong guarantees that the code loaded by the remote boot is correct, if the boot protocol says it is. The removal of maintenance and other attacks upon system integrity then becomes desirable in order to improve performance, rather than as a pre-requisite for ensuring correct behaviour.

Our approach makes essential use of a hash function which is deliberately chosen so as to be rich in collisions, in contrast with prevailing practice.

1. Introduction and Summary. We consider 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[1]. By relatively stateless, we mean that the workstation, while unattended, is not able to preserve with any degree of reliability the integrity of any mutable data (ie data which can potentially be changed without replacing the hardware.) In particular, we cannot rely upon the workstation to preserve the integrity of a secret such as a password or cryptographic key, since a secret is potentially subject to compromise and hence is liable to change. We also assume that the initiating user, being human, cannot reliably recognise or

[1] In the present context, this means that 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.

remember a well chosen (ie high entropy) key, whether completely private, shared secret, or public.

On the positive side, we do assume that the user can remember a poorly chosen (ie low entropy) password, that the user can force the workstation into a known initial state at will, that the workstation CPU and memory hardware can be trusted by the user to act correctly[2], and that the workstation can maintain the integrity of a small amount of immutable code (including a hash function and a driver for a secure keyboard), and hence can preserve the secrecy of a password entered from the keyboard (by forgetting the password prior to transferring control to any mutable code, including boot code.)

By a potentially hostile environment we mean that we make no assumptions about the integrity of the rest of the system. We allow the network to which the workstation is attached, and all the services accessed across the network, to be subject to passive or active 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, including such things as network device drivers.

We show how to perform a secure boot under these conditions. Our approach makes essential use of a hash function which is deliberately chosen so as to be rich in collisions. This contrasts with the practice of constructing hash functions so as to be as free as possible from collisions, which has been the historical practice when secrecy rather than integrity is at stake.

2. The Problem. Consider the following problem. We have a number of workstations, possibly in a public area[3], each of which may be used by a number of different, and possibly mutually suspicious, users. Each user has an operating system kernel (which may include access to a security policy service) which they trust for certain purposes, and which the user wishes to be certain is running on any workstation which they are using for as long as they are using it for that purpose. But there need be no global kernel which is trusted by all users for all purposes, or indeed by all users for any single purpose, and even a single user may require to use a number of different kernels, which are trusted by that user for different purposes.

The workstations are assumed to be tamperproof and initializable, in the sense that the keyboard, CPU, RAM and ROM hardware all functions correctly[4] , the initial contents of the ROM cannot be changed (neither by the user nor by anybody else under any circumstances) and there is a conceptual large red button which the user can press which has the effect of setting the program counter to a fixed address in ROM and disabling any other interrupts or untrusted hardware components such as DMA communications and storage devices.

[2]Ie that the user has good reason while using the workstation to believe that the right hardware is in the workstation, and is prepared to bear the risk of the right hardware acting wrongly. For the purpose of this paper we regard the problems of ensuring physical hardware integrity as separate from those of ensuring software integrity.

[3]A large number of locations must be regarded as effectively public for this purpose, including most people's homes and offices.

[4]"Functions correctly" is here used as a euphemism for "functions the way it did when the manufacturer tested it".

If there were a legitimate way of changing the contents of ROM, then there would be a potential attack corresponding to misuse of the change method. Apart from convenience of maintenance[5], our assumption that the ROM cannot ever be changed provides automatic protection against any such attack. We shall show below that this apparently very restrictive tamper proofing assumption need not actually cost anything in the way of convenience, whereas it does allow us to guarantee the user a high level of integrity.[6]

We assume that the kernel which a particular user wishes to use cannot be placed in ROM. This could be because the kernel is too big to fit, or because all the kernels available to be used by all the users are collectively too big to fit, or because the user wishes to be free to change to using a new kernel (or a new version of an old kernel) without altering the ROM (which by our assumptions would require alteration of the hardware.)

Therefore the workstation must download the kernel from somewhere else, for instance from a boot service accessed across a network to which the workstation is attached. How can we ensure that the code which we download is a true copy of the code which the user wishes to use?

3. Solution Strategies. The most obvious approach to the problem involves securing the entire network including all the boot servers. But even this would require the network addresses of all possible boot servers to be hardwired immutably into ROM at start of day. We shall show that this approach is not only impractical, but unnecessary.

An alternative is to adopt an end-to-end approach ([5]) and try to check that the code loaded (by whatever means) is correct prior to passing control to it. As we shall see, this approach has the added benefit that the user does not need to trust the boot code (including drivers) responsible for downloading the kernel, which can therefore be placed in RAM and changed (maintained) at will.

The simplest way of realizing this approach is to precalculate a checksum (hash) of the correct kernel code, and to check that the loaded code has the correct hash. But how can we be sure at boot time what the correct checksum is? If the checksum is long enough and sufficiently collision free to provide a strong guarantee of integrity, then by our assumptions neither the user nor the workstation can be relied upon to remember 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[7]. And it is not safe to store and download the checksum with the kernel code, since the hash function is publicly available and an attacker could therefore modify the kernel and recalculate the appropriate hash value. We could require the hash value to be signed by the party responsible for maintaining that particular kernel, for example using a public-key cryptographic system such as RSA, but neither the user nor the workstation can be relied upon to remember the appropriate public key, since this key has high entropy and will change abruptly if the keyholder believes their private key to have been compromised.

[5]There may be thousands of workstations.

[6]Provided that whatever protects the workstations from theft also protects their hardware from modification. This was not at one stage the case in the public area of the Computer Laboratory at Cambridge University: the anti-theft device allowed access to the workstation processor and students were able to insert an emulator which captured other users' passwords.

[7]It may be that the only such party is the user.

Some security could be provided by using a poorly chosen (low entropy) unshared secret, which we call a password, and which we assume is known only to a single user, and is used by them only for this purpose. The user can employ this password to maintain the integrity of a piece of data to be downloaded (data such as a checksum or a public key) as follows. First the user must obtain by some means an authentic copy of the data to be protected against modification. Next, the password is hashed together with the data in some way to produce a checksum. (For example, the password could be prepended to the data and the result hashed to give the checksum. Alternatively, the data could be hashed and then the hash encrypted, using the password as a secret key, to give the checksum.) Finally, this checksum is appended to the data. At the time when the data is downloaded, if the checksum calculated using the password matches that appended to the data, then there is a high probability that either the password is compromised or the data is unmodified.

Unfortunately, the first of these possibilities is quite likely if we use a conventional collision free hash function. A determined opponent can make an offline guessing attack by downloading the data and 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 exhaustive search within the computational resources of the opponent is likely to succeed. 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 so as to provide a large number of collisions. The idea is to ensure that an exhaustive offline search by the opponent will produce not one, but a largish number of candidate passwords, any one of which will produce the correct checksum for the correct data, but only one of which will produce the checksum that will be expected by the user for the bogus data as modified by the opponent. (A similar approach is applied to a different problem in [2].)

As we shall see, the effect of this strategy is to ensure that any guessing attack by the opponent is effectively forced online, in the sense that the attack now requires the user's interactive participation, and hence allows the user to offer a defence, which an offline attack does not.

4. Collision Rich Hashing. Before giving the details of a secure booting protocol, we show one way of constructing an appropriate collision-rich hash function from a conventional collision free hash function.

Suppose that h is a collision free one way hash function. Then the hash function defined by

$$q(k, x) = h((h(k|x) \bmod 2^m)|x),$$

where | denotes concatenation, will have the properties which we require, provided m is suitably chosen and h has suitable mixing properties (see [1] for details.) It is the reduction modulo 2^m which generates the deliberate collisions. We consider the choice of an appropriate value for m in Section 5.

So suppose that x is the data whose integrity the user wishes to protect, that k is the

user's password, and that the checksum $q(k, x)$ is appended to x. Now the attacker is faced with a dilemma. The attacker wishes to modify the data in some way, and then construct a checksum for the modified data which 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 abandon the attack or guess the password (which is at least a better bet than guessing the checksum directly.) But if the attacker guesses wrongly then the user will become aware of the attack. Of course, the user may wrongly attribute the mismatch of the checksum to a network error, or to a dirty sector on the boot server disk. But if the data x is followed by both $q_1 = (h(k|x) \bmod 2^m)$ and $q_2 = h(q_1|x) = q(k, x)$, then the user can almost certainly tell the difference between chance and deliberate attack.

If $q_1 \neq (h(k|x) \bmod 2^m)$ but $q_2 = h(q_1|x)$ then an attacker is almost certainly at work. In order not to allow the attacker to obtain the user's password by repeated guessing, the user should change password immediately upon detecting an attack of this kind. This cannot be done from the workstation being booted, for a number of reasons, many of them obvious. Consequently, as with most defences against an on-line penetration attack, there is now a risk of a denial of service attack, where the attacker deliberately corrupts the value of q_1 and re-calculates q_2 in order to prevent the user from using any of the workstations, possibly in the hope that the user will eventually respond by ignoring the integrity failure and proceeding regardless. But this is of course precisely what the user should do in any case, in order to ensure that the attacker gains no information about the correctness of the guessed password. The point is that the user is now aware that an attack is being made, and so knows not to continue to rely upon the integrity of the data.

As well as changing password upon detecting an attack, the user must also change password whenever the protected data changes, because otherwise the attacker will have two independent pieces of information about the password, which will reduce the number of possibilities for k revealed by exhaustive search to a dangerously small value[8]. Because of this, the burden on the user is less if the data protected directly by the password changes rather infrequently. Rather than use the password directly to protect the integrity of mutable data, it is therefore better to hash the data with a collision free hash function, and sign the hash with a (high entropy) private key. The password can then be used to protect the integrity of the corresponding public key (and of any cryptographic code necessary to verify the signature.) To ensure that the mutable data is fresh, a datestamp should be appended prior to hashing.

5. A Secure Boot Protocol. Now we apply the ideas which we have developed to give an example of a secure boot protocol for a relatively stateless workstation.

The user approaches a workstation and executes some untrusted local boot code. In response to a prompt, the user indicates that the secure boot protocol is being initiated, and which kernel it is desired to load. The untrusted boot code may load device drivers and various other bits of software from untrusted sources into the workstation before accessing the secure boot service via the network. As a result of execution, the local boot code may or may not correctly load into the workstation RAM the following:

1. certification code (described below), including code to perform public key cryptog-

[8]Typically one.

raphy;

2. the public key of the authority which the user regards as responsible for maintaining the kernel;

3. the two hash values q_1 and q_2 as defined in Section 4, taken over the concatenation of the data in (1) and (2);

4. the code for the secure kernel;

5. a certificate for the kernel, which consists of

   (i) an identifier for the kernel

   (ii) the value of a collision free hash function h applied to the kernel code in (4) and

   (iii) a datestamp (for freshness),

   all signed under the private key corresponding to (2).

The user now presses the red button to pass control to the immutable ROM code, and then inputs the password. The ROM code computes the 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 has been loaded, and whether the datestamp is acceptable. If all is well, then the certification code will pass control to the kernel.

Of course, the public key in (2) could be the user's own public key, which would allow the user complete control over the certificate (5). Any system code or device drivers loaded during the preliminary local boot and which are required to have integrity following the secure kernel boot need not be re-loaded, but can simply be included in the kernel checksum in field (ii) of (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 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 which contains (i) the kernel identifier (ii) the public key of the appropriate authority for that kernel and (iii) a datestamp, all signed under the user's private key corresponding to the public key in (2). This use of self-authenticating proxies is developed in [4]. Heterogeneity of hardware among workstations can also be accommodated in this way.

We conclude this section by discussing the choice of password. A number of tools are available which will generate, from a uniform distribution, a sequence of syllables which looks and sounds like an english word, but isn't. For example, the Concept Laboratories password generator [3] if asked to generate a twelve letter password will give one with an effective entropy of slightly over 28 bits[9]. Assuming that the user password k is generated in some 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, ie values in the domain for k which 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 and guess directly the correct value of $h(k|x') \bmod 2^m$ for the modified data x' and so deduce the values of q_1 and q_2 which 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.

6. Discussion and Conclusion. 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 can then be very sure that the workstation is in the same state as if the user had correctly hand typed the kernel into the workstation. Although some users will doubtless continue to prefer the second option, it is pleasant to have the choice.

It is perhaps worth pointing out that our approach to secure booting makes a complete separation of secrecy (read protection) and integrity (write protection). We require ROM code to be protected against tampering (unauthorized writing) but allow this to be done by not permitting any writing to it at all (immutability), and by making the code publicly visible (no read protection.) Similarly, we require the user's password to be kept secret (protected against unauthorized reading) but allow this to be done by not permitting it to be read at all by mutable code, and by forgetting (erasing or mangling) the password after a very short time (the opposite of 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.

References.

[1] L. Gong, T. Bergson and M. Lomas, 1994, Secure, Keyed and Collisionful Hash Functions, EUROCRYPT'94, *to appear.*

[2] L. Gong, T.M.A. Lomas, R.M. Needham and J.H. Saltzer, 1993, Protecting Poorly Chosen Secrets from Guessing Attacks, IEEE Journal on Selected Areas in Communication, **11**(5) 648–656.

[9] 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 then some choices are then more likely than others. In either case, the effective entropy is considerably less than the bit-length of the password.

[3] J. Gordon, 1993, Password Generation Software, Concept Laboratories, Lynfield House, Datchworth Green, Hertfordshire, England

[4] M.R. Low and B. Christianson, 1994, A Technique for Authentication, Access Control and Resource Management in Open Distributed Systems, Electronics Letters *to appear.*

[5] J.H. Saltzer, D.P. Reed and D. Clark, 1984, End-to-End Arguments in System Design, ACM Transactions on Computer Systems **2**(4) 277–288