Accountable System Administration Through Integrity Reporting

No author given

No institute given

Abstract. System administrators are, by necessity, some of the most trusted people in an organisation. Unfortunately, the administrator of a remote service platform is usually unknown and therefore difficult to gain trust in. We propose that this problem can be solved if platforms attest to the *administrative actions* carried out rather than just the software they are running. We describe an example of how this could be realised through the implementation of an integrity-measuring menu-based Unix shell. To manage the log of attested actions, we also introduce a generalised framework based on process algebra for verifying integrity measurement logs.

Keywords: Attestation, System administration, Integrity reporting, Process algebra

1 Introduction

The increasing popularity of service-oriented architectures and cloud computing means that more data are being stored and processed on remotely administered platforms. Service *providers* are therefore becoming more trusted. This may be reasonable: a remote service provider is likely to be contractually obliged to keep data in a secure manner and severe penalties may apply should anything go wrong. However, this level of assurance is not always enough. If the data is worth more than the contract, or is particularly commercially sensitive, then the threat of an insider attack remains. Furthermore, while the company may be trusted, the individual system administrators may be motivated to compromise the service platform and steal valuable intellectual property or alter the results of computation. One example of this is when processing confidential or controversial scientific data – such as research on pharmaceuticals or climate change – significant financial rewards may exist to copy or alter the results. As a result, methods of assurance that *do not assume that the system administrator is trustworthy* are required.

Attestation, as defined by the Trusted Computing Group, can potentially provide relying parties with a high level of assurance in the integrity of remote software platforms. However, it is generally accepted that attestation does not report enough information to trust the *behaviour* of those platforms [1]. This is partly because most software can be used for both good and bad purposes,

and merely knowing the identity and integrity of that software is insufficient to trust it. There is a significant part of the system's execution context missing: the administrators and users of the platform who may alter the behaviour of its software in undesirable ways. As a result, we propose to extend integrity reporting to attest to *administrative actions*, demonstrating that the administrators are behaving in a trustworthy manner with securely identified and trusted software.

In doing so we address three key problems. Firstly, the practical problems of remote attestation [2,3] and the overhead of managing many configurations of software [4]. Secondly, the fact that system administrators must have privileged accounts on the system in order to do their job, and that an assurance system must allow for emergency actions to be taken. Thirdly, the task of verifying potentially complex attestations of platform state.

The paper is structured as follows. In section 2 we introduce trusted computing concepts as well as the basis of our solution: a menu-based Unix shell. A more detailed scenario and threat model are covered in section 3. Section 4 describes our proposed method for attesting administrative actions and section 5 introduces a new integrity verification approach using the CSP process algebra. In section 6 the solutions are analysed to see how well they address the identified security issues and threats. Section 7 covers related work and several discussion points and in section 8 we conclude.

2 Background

2.1 Trusted computing

Trusted computing is a paradigm developed and standardised by the Trusted Computing Group [5]. It aims to enforce trustworthy behaviour of computing platforms by identifying a complete 'chain of trust', a list of all hardware and software that has been used. If a platform owner can reliably find out exactly what software and hardware is in use, they should be able to recognise and eliminate any malware, viruses and trojans. A great deal of infrastructure is required to make this idea practical, including new hardware, modifications to applications and databases of known, trustworthy platform configurations.

The technologies proposed by the TCG are centred on the Trusted Platform Module (TPM). In a basic server implementation, the TPM is a chip connected to the CPU. It provides isolated storage of RSA keys and Platform Configuration Registers (PCRs). PCRs can be used to hold *integrity measurements*, in the form of 20 byte SHA-1 hashes. They can only be written to in one way: through the `extend(..)` command. This appends the current register value to the supplied input, hashes it, and stores the result in the PCR. A PCR value therefore reflects a *list* of hashes. In order to work out what individual inputs have been added to a PCR, a separate log must be kept. When this log is replayed, by rehashing every entry in order, that final result should match the PCR value.

The limited functionality offered by the TPM is ideal for recording the boot process of a platform. The idea being that, starting from the BIOS, every piece

of code to be executed is first hashed and extended ('measured') into a PCR by the preceding piece of code. This principle is known as *measure before load* and must be followed by all applications. If so, no program can be executed before being measured, and because the PCRs cannot be erased, this means that no program can conceal its execution from the TPM. The first module cannot be measured, and is referred to as the *root of trust for measurement*. A platform is said to support *authenticated boot* when it follows this process as it provides a way for the platform's boot process to be authenticated against reference values at a later time.

Authenticated boot can be extended to the application level through an integrity-measuring operating system. IMA Linux [6] is a Linux Security Module which can measure all executed applications and shared libraries (as well as other files) into PCRs.

2.2 Remote attestation

In order for users to assess a remote machine, the TPM supports *remote attestation*, a process that allows a platform to report the integrity measurements collected during authenticated boot. When challenged, the TPM can create a signed copy of its PCR values. This is given to the challenger for inspection, along with the measurement log. The PCRs are signed using a private key held by the TPM, guaranteeing the key's confidentiality. This is called an Attestation Identity Key (AIK) and the public half must be certified by a certificate authority (a 'Privacy CA'). Full details are on the TCG website [5].

In addition to PCRs, TPMs also contain at least four monotonic counters. These counters can only be incremented and may be used to count the number of times a platform has booted. Counter value can be attested through creating a log of a TPM transport session where a `TPM_ReadCounter` operation is called, and signing it using an AIK [7].

2.3 Protecting data and keys

The TPM can encrypt arbitrary data so that it can only be decrypted when its PCRs have certain pre-defined, trustworthy values. This can be used, for example, to prevent an unauthorised operating system from accessing a private SSL key. One way this can be implemented is by creating a TPM key which is *bound* to the PCR values through the `CreateWrapKey` command. The private half of the key is then always held securely in the TPM. When it needs to be used, a request ('`unseal`' or '`unbind`') is made to apply the private key to the encrypted data. The TPM will only complete the request when the PCRs are in the state defined upon key creation.

2.4 PDMenu: a menu-based Unix shell

Menu-based shells are alternatives to UNIX command-line shells such as BASH. Instead of providing a command interpreter with free-form input, they limit the

user to running commands given in the menu. A menu-based shell can be set as the user's default shell in `/etc/passwd` so that it will run whenever the user logs in or opens a terminal. The security of a menu-based shell is largely determined by the menu options it allows, as well as the quality of the implementation. Should any of the menu items allow launching another application then the security benefit of a menu-based shell is lost. The general principle of the menu-based shell is very similar to a *secure boot* [8] concept in trusted computing: the user is limited in what they are allowed to run, but that may not necessarily be trustworthy.

The PDMenu [9] shell is an example of a simple menu-based shell. It presents users with a menu containing options specified in the `.pdmenurc` file. The PDMenu configuration file is powerful, but we will only be concerned with the *exec* option which creates a menu item for running an application. An example of a menu configuration is shown in figure 2. Each *exec* line takes the form:

```
exec:description:flags:command
```

The `description` section is the menu option that will be displayed to the user when PDMenu starts. The `flags` are details of how to run the command. The command itself is the string passed to the `system(...)` function and executed.

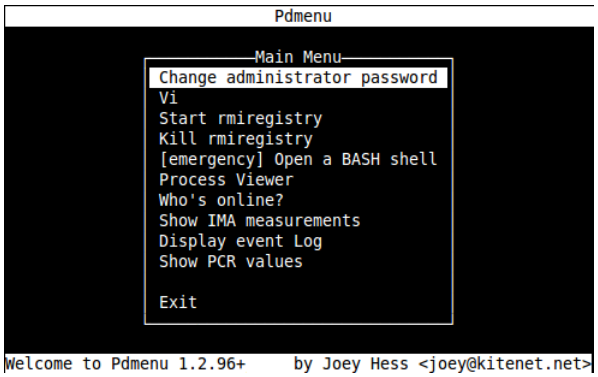

Fig. 1. An example instance of PDMenu.

3 Scenario and Threat Model

We imagine a scenario based on administration of a single remote server. This may be a web service or a cloud computing instance. The users of the server have high confidentiality and integrity requirements. They may be running scientific simulations for controversial research, have valuable intellectual property or be processing financial data. The main assets requiring protection are the

```
menu:main:Main Menu
  exec:Change administrator password:p:passwd
  exec:Vi::vi
  exec:Start rmiregistry:p:rmiregistry &
  exec:Kill rmiregistry:p:killall rmiregistry
  exec:[emergency] Open a BASH shell::
       ./jtt.sh pcr_extend -f emergency.txt -p 12; bash
  exec:Process Viewer:truncate:ps aux
  exec:Who's online?:truncate:echo "These users are online:";w
  exec:Show IMA measurements:display:cat /sys/kernel/security/ima/ascii*
  exec:Display event Log:display:cat ./menulog.txt
  exec:Show PCR values:truncate:./jtt.sh pcr_read
  exit:Exit
```

Fig. 2. An example PDMenu terminal configuration file

programs and the data being used and produced. For cost or availability reasons the server is being administered by a remote provider. There may be many administrators and they are relied upon to do a combination of basic maintenance: running backups or restoring from them, adding or remove users, starting or stopping jobs, changing passwords, installing patches, and so on. However, their most important task is rescuing the system when something goes wrong: when a process is taking too much CPU and needs to be killed, for example. An equally valid scenario would be management of a particular individual application via an administrative interface.

The users of the remote server do not want to have to trust every administrator. They trust the software that is being run – either it was developed by the users or they have performed a thorough audit – but have no way of assessing the honesty of the administrator. They also have concerns about malware such as rootkits and therefore challenge the server to attest regularly. The server must therefore be using authenticated boot and be running an integrity-measuring operating system. Furthermore, they require the server to be available as often possible and every attestation failure will be expensive for their day-to-day operations.

There are many threats we do not consider in the paper. Hardware attacks are ignored, as these can be mitigated through insisting the hardware is provided by a third-party infrastructure (IAAS) provider. Runtime attacks by a remote party are out of scope for this paper, but remain a problem. For this reason, the solution we propose has a small code footprint to minimise the chance of a runtime vulnerability being exploited. We assume that secure, TPM-backed storage is implemented so that a rogue administrator cannot reboot the server into an untrustworthy operating system or copy the hard drive contents.

The threat that we seek to minimise is that of a rogue administrator using privileged access to either steal confidential data or modify the running appli-

cations. However, the administrator is assumed to have (and need) command-line access to the server, as well as sometimes requiring the ability to run as a super-user. We assume that they do not have access to any other administration interfaces. The next section proposes a compromise solution which allows a certain amount of system administration while restricting access and providing *attestable* audits of every action.

4 Using *TPDMenu* to Attest to Administrative Actions

Providing system administration *and* attestable assurance of the platform despite a potentially untrustworthy administrator appears to be a fundamentally impossible task. If the administrator is given root access to a command line, for example, they would be able to perform any number of attacks. However they may need this capability to fix important issues.

One obvious solution is to provide accountability for administrative tasks. If all actions the administrator takes are logged, and that log cannot be tampered with, then users could at least know when their system has been attacked. This would discourage potential inside attackers. One way to do this, as suggested by Sailer et al. [6], is to use an integrity-measuring operating system and shell. Every executable and command run in the shell is extended into a PCR, providing the desired accountability as the system can now attest to what the administrator has done. However, this has two major drawbacks. Firstly, this may be too limited a solution as the damage may already have been done. Secondly and perhaps more importantly, the overhead on the challenger is enormous. Not only must they interpret the attestation, they must be able to identify malicious behaviour from lines run in a command shell. These could be obfuscated in any number of ways making this impractical. Almost every tool that the administrator might use can probably be used for untrustworthy reasons. Simple integrity measurement of commands is not a viable solution.

Instead, the range of commands that the administrator is able to perform must be limited to a reasonable number. To do this we propose the use of a customised *menu-based shell*. As described in section 2.4 the shell presents the administrator with a selection of menu items for actions he or she is allowed to perform. In the example in figure 1, the administrator is limited to only a few options, including changing their password, opening an editor, starting or stopping the rmiregistry (a Java service required for remote method invocation), opening a BASH shell and a few other functions. When an option is selected the relevant executable is run and, on completion, the user is returned to the menu-based shell. The menu is configured using a simple configuration file, the one used in this case is shown in figure 2. This configuration file is effectively a *white list* of allowed administrator actions.

To prove that the administrator is limited to this shell a number of changes are required in the operating system of the server. We assume that PCR integrity measurement is supported through a system such as IMA Linux [6]. The kernel must be capable of measuring when a user logs in, which the IMA system does

already through measurement of the PAM authentication module and/or SSHD service. Next, the kernel's login program must measure the name of the user and the `/etc/passwd` file. This shows that the logged-in user is running the menu-based shell. The shell must then be measured along with the configuration file itself. Assuming the OS and other software is trusted and that it is not possible to break out of this shell or modify the boot sequence, then attestation should provide sufficient evidence that the administrator can only use this shell.

4.1 Implementation

We developed our prototype, *TPDMenu*, by customising the open source PDMenu shell [9]. The shell was modified to measure the configuration file it loads into PCR 10. We then modified menu execution to make PDMenu measure the *command string* executed at runtime into PCR 11. We allow the IMA system to measure the actual executable. This means that a verifying party knows the exact command executed and the hash of the binary.

In order to protect the data stored on the server we suggest using sealed TPM keys. A key can be created sealed to the server's PCR values *except* for PCR 11, the one extended with administrator actions. However, the key *must* be sealed to all binaries and the menu configuration file. Doing this in a reliable way relies upon pre-measurement of binaries as suggested by Kyle and Brustoloni [10] which should be straight-forward to implement and is easily verifiable. The same key can also be sealed to the server's configuration *before* anyone attempts to log into the shell. These sealed keys allow the limited administrator actions without loss of availability.

An alternative, equivalent option to pre-measurement of binaries would be a *secure boot* menu which is pre-loaded with the expected hashes of the executables it will launch and checks these before actually running them.

The original shell consists of nearly 2000 lines of C code[1]. Our modifications introduced fewer than 200 more. We could reduce the total significantly by removing the code we had considered unsafe and disabled. We believe this makes it a reasonable size for auditing and therefore sensible to potentially trust. The menu files are small and could easily be analysed by an external party.

4.2 'Break-the-glass' policies

Menu-based shells can also be used to implement 'break-the-glass' policies [11]. These allow emergency actions to be performed so long as they are reported or result in additional constraints. The menu item given on line six of figure 2 is an example of this: '`exec:[emergency] Open a BASH shell`'. A bash terminal can be opened, but results in another PCR, number 12, being extended. The terminal can also extend all commands into the PCR for auditing. This will be reported in attestations. Any 'top secret' data can be sealed to PCR 12 in order to prevent an administrator from 'breaking the glass' and then making an unauthorised

[1] All line counts generated using David A. Wheeler's 'SLOCCount'.

copy. This provides the balance between availability and protection of the most critical data. Other actions might be necessary, such as deleting local caches or stopping processes which are still using confidential data. The key advantage of this approach is that remote parties can be sure that these will be carried out before super-user privileges are granted. Of course, to recover access to this data the system must be restarted. Future work will investigate solving this problem.

An alternative way to provide this functionality is to allow login to the system with another username who is configured in `/etc/passwd` to use the bash prompt. Again, this would require a customisation to measure a value into PCR 12, but would otherwise provide the same functionality.

Measured item	PCR	Data sealed
Boot process and TPDMenu	0-10	Secret and top secret user data
TDPMenu admin events	11	None
'break the glass' events	12	Top secret user data

Table 1. Protecting confidential data

4.3 Protecting against system reboots

A malicious system administrator attempting to cover his or her actions might reboot the platform such that the evidence (as stored in PCRs) is removed. Although the use of sealed storage will protect data despite this, it may be important to know that an administrator temporarily put the platform into an untrustworthy state. One way of achieving this is through use of a TPM monotonic counter. Early in the boot process the counter is incremented. When shutting down, the platform creates an attestation of its counter and PCR values and saves these along with the integrity measurement log to persistent storage. This record can be exported later to relying parties. Improperly shutting down the platform will not create this log and will therefore alert another administrator (or users) that something bad may have occurred.

However, attempting to forge the shutdown record might be possible (e.g., by creating the log and then aborting the shutdown process). This can be avoided by extending random values to all PCRs before creating the log and therefore removing access to *all* sealed data after a reboot has been requested. We leave a full solution to this problem as future work.

5 Integrity Verification with CSP

The implementation described is relatively straight-forward to verify. Standard integrity verification is used to check the first 8 PCR values against trusted ones.

This is followed by verification of the login state (has an administrator logged in?), other executables, TPDMenu shell binary hash and menu configuration file hash in PCR 9 and 10. Finally, PCR 12 can be checked for any emergency administration actions and PCR 11 can be analysed to see what actions have occurred. Assuming data is sealed in the manner described previously then this is sufficient.

However, when PCRs are being used for both binary measurements and *run-time events* – the system administrator's actions – verification has the potential to become complicated. Other systems such as [10,12,13,14] take a similar approach and rely on increasingly complicated verification processes. For this reason we propose a sophisticated *general* approach to integrity verification using process algebra. The goal of this method is to provide a general framework for integrity verification of *any* system.

5.1 CSP

Communicating Sequential Processes (CSP) is a process algebra commonly used to describe interacting concurrent systems. A full explanation of CSP can be found in [15]. Processes are defined by name and make a series of communications before either stopping (taking the behaviour of process 'STOP' which is defined as a process that never communicates) or behaving like another process. For example, process P communicates a and then behaves like Q. Process Q communicates b and then never communicates again:

$$P = a \rightarrow Q$$
$$Q = b \rightarrow STOP$$

Arrows ($\rightarrow$) show the sequence of events. Processes may be composed in parallel and must synchronise on any communications they share. These are defined explicitly in the composition, for example $P \llbracket a, b \rrbracket Q$ shows process P and Q must synchronise on communications a and b. External choice is shown with a square ($\Box$).

Messages can be communicated between processes through channels which have *inputs* and *outputs*. In the figures, inputs are shown with a question mark and outputs are shown with an exclamation mark. For example, the TPM process in figure 6 shows the TPM waiting to synchronise on channel *extend*, where it receives a message into object x and then outputs the same value on channel *tpmextend*. Multiple input and outputs are separated using the same question mark, dot or exclamation mark. If a specific message is defined for the input, the process will only synchronise if the right output on that channel is given by another process. For example, the $PCRLOG1$ process in figure 4 synchronises on two values, the first of which must be 1 and the second must be *bios*.

5.2 Approach

We have used the CSP language to model a platform and describe how it interacts with trusted computing components. We propose that all measured executables have a representation as processes in CSP and that a complete system is simply a composition of these processes. Each process is defined in terms of what it extends to each PCR and how it passes control to other processes. A system model can be seen in figure 6 which shows the TPM and a simplified boot process including the BIOS, boot loader and IMA Linux operating system. An overview of the entire process is given in figure 3.

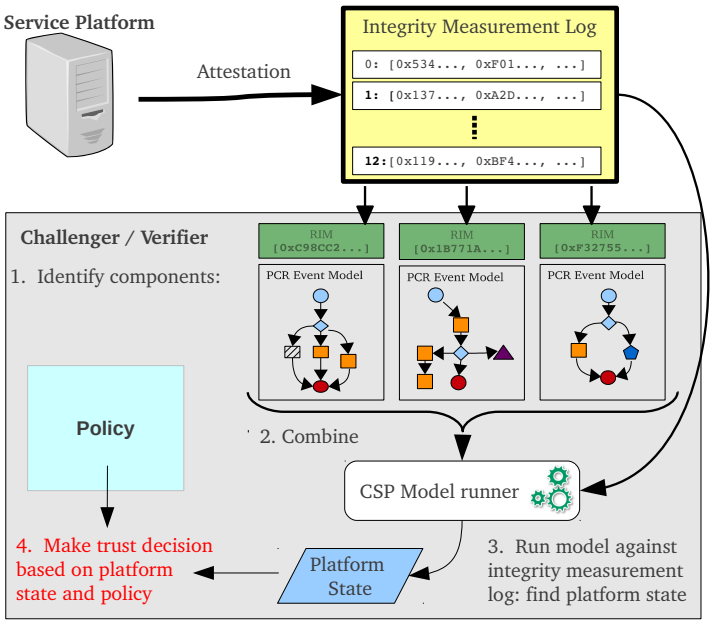

Fig. 3. An overview of the CSP IML verification approach

Assuming that all system components are defined using CSP, the FDR2 tool [16] can be used to check a system model against a sequence of TPM extend actions – the measurement log – through *trace refinement*. Trace refinement [15] can confirm that a measurement log could have been produced by the system model. An example of this process can be seen in figure 4 where the SYSTEMH process (defined in figure 6) is checked against the PCR logs.

We cannot say for certain that this system model is accurate, only that it is a plausible explanation for the integrity measurement log. As such, we rely on the software to work in the way that its model describes, and on the authenticate boot process guaranteeing that no other software has been executed. We also rely on the composition of these models (and any interference or incompatibility

between them) being accurately described. With these assumptions, we believe this process to be sound.

Having established that the measurement log could have been produced by a system conforming to the CSP model, the verifier can then inspect the resulting system state to identify how it may behave in the future. One approach is to reason about (in CSP terms) the 'afters' of the trace: what the model can still do. This is useful for establishing what possible events could occur, for example, after an administrative intervention, will the platform still accept user input? Furthermore, in the case of 'break-the-glass' policies, it is possible to show that any potentially unsafe action must occur *after* a specific PCR has been extended.

However, the process models describe only how the system will behave with respect to PCR usage. It is perhaps more sensible, therefore, to stop after obtaining the system state and examine the remaining processes that are, according to the model, currently running or able to run. The method for doing this will depend on the verifier's security policy and is out of scope of this paper. In situations where PCRs are used to measure specific events, however, the verifier can go further and identify the *internal* state of a particular process and whether it can be trusted. Again, this will depend on the process and security policy, but this approach provides a standard way of describing the system state.

5.3 Example

To demonstrate how this approach might be used, a basic model of the TPDMenu shell is given in figure 5. For clarity, all applications are shown extending their own name rather than an application hash, e.g., `extendreq!tpdmenu` . In this model the `SHELL` process starts and then launches the `TPDMENU` process, which offers two menu options: `who` and `ps`, each of which corresponds to other application models which have been omitted.

Figure 6 shows process `SYSTEMH`, which composes together a platform boot process (`BIOS`, `BOOTLOADER`, `IMA`, `TTY`) with a TPM and applications (`APPS`). The `BIOS` starts by extending its own measurement to PCR 1, then extending the bootloader to PCR 2 and then transferring execution to the `BOOTLOADER` process. `BOOTLOADER` extends PCR 7 with the `IMA` measurement, in this case representing the whole of IMA Linux. `IMA` then loops indefinitely, waiting for communications either through the `launchreq` channel, which is for executing new applications and extending their hash to PCR 10 and the `extendreq` channel, which is for extending arbitrary values to PCR 11. Caching is supported such that PCR 10 is only extended once per application. Applications, such as the `TPDMENU` process in figure 5 must be prefixed with a communication waiting to synchronise with the operating system, e.g. `launch.tpdmenu` so that they cannot start until the IMA process has had the chance to be extended into a PCR.

LogVerificationExample

channel
$extend, tpmextend, finishextend$

process
$PCRLOG1 = tpmextend.1.bios \rightarrow ...$

process
$PCRLOG2 = tpmextend.2.bootloader \rightarrow ...$

assert
$SYSTEMH \sqsubseteq_t (PCRLOG1 \;|||\; PCRLOG2...)$

Fig. 4. CSP log verification example

TPDMenu

process
$SHELL = launch.shell \rightarrow launchreq!tpdmenu \rightarrow STOP$

process
$TPDMENU = launch.tpdmenu \rightarrow TPDMCONF$

process
$TPDMCONF = extendreq!11!who \rightarrow hasextended.who \rightarrow launchreq!who \rightarrow TPDMCONF \;\Box$
$extendreq!11!ps \rightarrow hasextended.ps \rightarrow launchreq!ps \rightarrow TPDMCONF \;\Box$
$TPDMENU$

Fig. 5. CSP model of the TPDMenu menu-based shell

6 Evaluation

6.1 TPDMenu Security Analysis

This section considers the security of a platform running TPDMenu against a malicious system administrator. The assets requiring protection are the data on the platform (confidentiality, integrity) and the software (integrity). The system administrator may attack in a number of ways: by logging in and reading files or memory state, simulating an emergency condition, executing a local exploit, or bypassing the shell and using a different interface.

Firstly, they might log into the platform and try to modify or read a file. This would only be possible if they can log into the TPDMenu shell and one of the options allows access to important files or a normal terminal. The presence of this option would be logged to a PCR and would not be trusted by a remote party. They would therefore not seal any data to this configuration or run software on this platform when it was in this state. The same system would prevent the administrator from booting an alternative OS or installing new software.

SystemBoot

```
channel
    extend, tpmextend, finishextend

channel
    channellaunch, launchreq, extendreq, hasextended, invalidate, hasinvalidated

process
    TPM = extend?p?x → tpmextend!p!x → finishextend!p!x → TPM

process
    BIOS = extend!1!bios → finishextend.1.bios →
           extend!2!bootloader → finishextend.2.bootloader → BOOTLOADER

process
    BOOTLOADER = extend!7!ima → finishextend.7.ima → IMA(⟨⟩)

process
    IMA(s) = launchreq?x → CACHE(s, x) □
             extendreq?x → extend!11!x → finishextend.11.x →
                 hasextended!x → IMA(s)

process
    CACHE(s, x) = if elem(x, s)
                  then launch!x → IMA(s)
                  else extend!10!x → finishextend.10.x → launch!x → IMA(s ⌢ seq x)

process
    SYSTEM = (TPM |[ extend, finishextend ]|
              (BIOS |[ launchreq, launch, extendreq, hasextended,
              invalidate, hasinvalidated ]| (APPS)))

process
    SYSTEMH = SYSTEM \
              {extend, finishextend, launch, launchreq, extendreq, hasextended,
                  invalidate, hasinvalidated}

process
    TTY = launchreq!shell → TTY
```

Fig. 6. CSP model of the TPM, platform boot process and IMA Linux

If there was an emergency requiring intervention by the administrator, this could be an opportunity to gain access to the system. Temporarily this would be the case: the administrator may need root access and could copy files or modify executables. However, as soon as this happened, the 'break-the-glass' rules would apply and some data could be made inaccessible. Furthermore, these emergency actions could still be logged. For the system's functionality to be fully trusted again the system must reboot. This will restore a known-good platform state.

However, the system administrator is now given the possibility of performing denial-of-service attacks on the platform. By claiming that an emergency action which requires extending PCR 12 is necessary, they can disable access to sealed data. This may be unacceptable in some situations but is still an improvement on systems which either provide full administrative access or require a reboot on any unusual intervention. A potential solution would be to allow remote users

to send a new version of the key to their sealed data to the new platform state should they deem the administrative action acceptable.

Local exploits remain possible, but are constrained by what actions can be carried out by the administrator. As the TPDMenu shell limits their input significantly, they must exploit an option that is considered 'trusted' by users. For example, in figure 1 the administrator could run `Vi` and gain access to a shell or find another flaw. The administrator might alternatively take advantage of an exploit remotely, but this issue is almost the same as an external attacker. We consider this a reasonable limitation.

Perhaps the most obvious attack is to bypass the TPDMenu shell and take advantage of some other interface. For example, through an FTP server or web interface. Again, this comes down to the trustworthiness of the rest of the software stack: we expect that users will only trust a system which is not running such programs. Breaking out of the TPDMenu shell might be possible. However, as mentioned earlier, the shell is extremely simple and it seems plausible to make this system attack-proof. The most common ways of breaking out of the shell otherwise would involve one of the allowed menu options or the exploit of an operating system bug. However, we think that the use of TPDMenu has at least reduced the attack surface.

In summary, TPDMenu limits the potential for most attacks that an administrator could perform, although it does rely on the trustworthiness of the rest of the software stack.

6.2 Practicality of administration with a menu-based shell

Administration of a complex system with a simple menu-driven system may not be practical. Some systems may not require any routine administrative actions, but will often need the superuser to have full root access and be able to issue arbitrary commands. In these systems, TPDMenu can only provide 'break the glass'-style policies.

However, we suggest that the same approach could be applied to many other administrative interfaces, such as network routers or web hosting control panels. For future work we would like to investigate the most appropriate context for using this approach and validate whether it is secure as well as usable.

7 Discussion and Related Work

There are several areas for further discussion, particularly in comparison to related work. The principle behind a menu-based shell is analogous to the execution of a trusted hypervisor from a list of known-trusted options in Intel TXT [17]. PRIMA [18] could also be considered similar, as it also constrains the platform using attestation of a trusted component: the SELinux security framework and policy. However, TPDMenu is more specific to the threat of an administrator, and is far less complicated an implementation. However, TPDMenu does not

allow the same flexibility as PRIMA for controlling untrusted applications and relies on the use of data sealing.

UCLinux [10] is the most closely related system, but focuses on usage control rather than system administration. From it we borrow TCB *pre-logging*. However, as an improvement, the TPDMenu system provides 'break-the-glass' functionality as well as administration without requiring system reboots.

Our approach to verification is novel, although some existing work touches on similar areas. Rohrmair [19] uses CSP to model platform start-up processes and the TPM. However, the focus is on verifying protocols rather than integrity measurement logs. Namiluko and Martin [20] use CSP to create an abstract model of a platform. Again, however, the focus is different. They aim to model platforms in sufficient detail to verify against architectural properties. This provides more detail but is significantly more complicated in comparison too our scheme. Naumann et al. [13] and Alam et al. [21] have developed a similar process for verifying behavioural updates on a platform. Their framework is intentionally abstract, and has been refined primarily for enforcing usage control, rather than the validation of reported attestations. Our approach based on CSP could be a useful alternative implementation of this model for more general integrity verification.

8 Conclusion

We have demonstrated that administrative actions can be captured and attested through TPDMenu, a menu-based UNIX shell. It takes advantage of platform configuration registers to make a limited amount of system administration possible without the end user having to completely trust the administrator. We have shown that it is simple to implement, requiring only limited modifications to the existing shell, and that it allows for potentially powerful security policies including 'break-the-glass' behaviour suitable for high-availability environments. The key realisation is that systems such as TPDMenu are easy to attest because they contain their own measurement white lists, in this case through the menu configuration file.

As a second contribution, we describe a general-purpose integrity measurement verification procedure using Communicating Sequential Processes (CSP). This allows for a model of platform state to be generated from system models and integrity measurements. CSP component models can provide a general-purpose, compositional method of describing any system that interacts with a TPM. There is a great deal of future work in this area, particularly in implementing policies to take advantage of the models that CSP generates. However, our main proposal for future work is the identification of further practical issues and improvements in a real deployment scenario. This will allow us to evaluate the cost and benefit with this approach in context.

References

1. Poritz, J.A.: Trust[ed | in] Computing, Signed Code and the Heat Death of the Internet. In: Proceedings of SAC '06, New York, NY, USA, ACM (2006) 1855–1859
2. Lyle, J., Martin, A.: On the Feasibility of Remote Attestation for Web Services. In: Proceedings of SecureCom '09. Volume 3., IEEE Computer Society (Sep 2009) 283–288
3. Coker, G., Guttman, J.D., Loscocco, P., Sheehy, J., Sniffen, B.T.: Attestation: Evidence and Trust. In: Proceedings of ICICS '08. Volume 5308 of LNCS., Springer (2008) 1–18
4. England, P.: Practical Techniques for Operating System Attestation. In: Proceedings of TRUST '08. Volume 4968/2008 of Lecture Notes in Computer Science., Villach, Austria, Springer Berlin/Heidelberg (March 2008) 1–13
5. The Trusted Computing Group: Website. `http://www.trustedcomputinggroup.org/` (2012)
6. Sailer, R., Zhang, X., Jaeger, T., van Doorn, L.: Design and Implementation of a TCG-based Integrity Measurement Architecture. In: USENIX Security Symposium. (2004) 223–238
7. Sarmenta, L.F.G., van Dijk, M., O'Donnell, C.W., Rhodes, J., Devadas, S.: Virtual monotonic counters and count-limited objects using a TPM without a trusted OS. In: STC '06: Proceedings of the first workshop on Scalable Trusted Computing, New York, NY, USA, ACM (2006) 27–42
8. Arbaugh, W.A., Farber, D.J., Smith, J.M.: A secure and reliable bootstrap architecture. In: Proceedings of the IEEE Symposium on Security and Privacy. SP '97, Washington, DC, USA, IEEE Computer Society (1997) 65–
9. Hess, J.: PDMenu Website. `http://kitenet.net/~joey/code/pdmenu/` (August 2009)
10. Kyle, D., Brustoloni, J.C.: Uclinux: a linux security module for trusted-computing-based usage controls enforcement. In: Proceedings of STC '07, ACM (2007) 63–70
11. Ferreira, A., Cruz-Correia, R., Antunes, L., Farinha, P., Oliveira-Palhares, E., Chadwick, D.W., Costa-Pereira, A.: How to Break Access Control in a Controlled Manner. In: Proceedings of CBMS '06, IEEE Computer Society (2006) 847–854
12. Gu, L., Ding, X., Deng, R.H., Xie, B., Mei, H.: Remote attestation on program execution. In: Proceedings of STC '08, ACM (2008) 11–20
13. Nauman, M., Alam, M., Zhang, X., Ali, T.: Remote Attestation of Attribute Updates and Information Flows in a UCON System. In: Proceedings of TRUST '09. Volume 5471 of LNCS., Springer (2009) 63–80
14. Lyle, J.: Trustable Remote Verification of Web Services. In: Proceedings of the TRUST '09. Volume 5471 of LNCS. (2009) 153–168
15. Roscoe, A.: The Theory and Practice of Concurrency. Prentice Hall (1998)
16. Formal Systems (Europe) Ltd: FDR2 User Manual. (1992) `http://www.fsel.com/fdr2_manual.html`.
17. Grawrock, D.: Dynamics of a Trusted Platform. Intel Press (February 2009)
18. Jaeger, T., Sailer, R., Shankar, U.: PRIMA: policy-reduced integrity measurement architecture. In: Proceedings of SACMAT '06. (2006) 19–28
19. Rohrmair, G.T.: Using CSP to Verify Security-Critical Applications. PhD thesis, University of Oxford (Hilary 2005)
20. Namiluko, C., Martin, A.: Abstract model of a trusted platform. In: Proceedings of INTRUST '10. (2010)
21. Alam, M., Zhang, X., Nauman, M., Ali, T., Seifert, J.P.: Model-based behavioral attestation. In: Proceedings of SACMAT '08, ACM (2008) 175–184