Of Headless User Accounts and Restricted Shells

JAN SCHAUMANN

Jan Schaumann is a Principal Paranoid at Yahoo!, a nice place to stay on the Internet, where he worries about scalable infrastructure and systems architecture. He is also a part-time instructor at Stevens Institute of Technology, where he teaches classes in system administration as well as in UNIX programming.
jschauma@netmeister.org

UNIX system accounts not bound to a particular user, so-called "headless user accounts," are frequently used to allow for automation of certain tasks. For security reasons, such headless accounts usually have a very restricted shell, allowing only a few select commands. At the same time, system administrators and service engineers frequently have a need to let such accounts execute additional commands, even though allowing an interactive shell is not an option. To address this problem, we developed a command interpreter called sigsh [1] that requires a cryptographic proof of authenticity and integrity (i.e., a "signature") by an authorized party before it executes a set of commands. sigsh(1) is currently used by Yahoo! Inc. on over a quarter-million hosts to help discover potential software vulnerabilities.

Systems engineers frequently make use of a headless account in order to, for example, automate the transfer files in and out of a host, perform certain asynchronous monitoring and reporting tasks, or run specific commands. In addition to possibly diluting the audit trail of any such actions, such accounts may pose a risk if access credentials (such as a public SSH key) are shared. To mitigate the risk of such an account being used in unauthorized ways, it is common practice to restrict the set of commands the account can execute to a select few. However, by restricting the set of allowed commands, usability is lost; people frequently need to be able to run additional commands not included in the restricted shell's internal whitelist. As a result, it is unfortunately not entirely uncommon for engineers to simply change the restricted shell to a fully interactive shell, proving once more, "The more secure you make something, the less secure it becomes" [2].

The conundrum posed can thus be described as the need to combine the conflicting requirements of unrestricted access for usability reasons with a restricted and individually authenticated way to execute commands: what's required is a way to allow arbitrary commands, provided they were sanctioned by somebody we trust. ("Arbitrary" here does not mean any random command, but, rather, any command not previously whitelisted.)

Account Types and Trusted Commands

In general, headless user accounts can be grouped into the following categories:

- system accounts provided by the operating system (OS)
- headless accounts with a completely disabled shell (such as /sbin/nologin or /usr/bin/false)
- headless accounts with a restricted shell
- headless account with an interactive shell

Let's take a close look at each of these cases, focusing on two core components in system security: data integrity and authenticity (the third main component, data confidentiality, is, in this context of command execution, irrelevant, although provided by the transport mechanism, that is, ssh/ssl) and the impact on the trustworthiness of the system as a whole.

System Accounts and Headless Accounts with Logins Disabled

The concept of system accounts and how they help implement the Principle of Least Privilege is assumed to be understood, so allow me to simply assert that these accounts are implicitly trusted to be running the given service (and any processes forked off it), but are not trusted to execute anything else. The most common examples for this type of account are the various system accounts used by some of the standard UNIX daemons such as named (commonly used by the DNS system), apache (commonly used by the Apache Web server) or sshd (used by the SSH service for the explicit purpose of privilege separation), to name but a few. With interactive logins explicitly disabled, they are included here purely for completeness' sake.

Headless Accounts with a Restricted Shell

Some headless accounts need to be able to run commands that are triggered asynchronously. That is, while it's possible to use these accounts to run scheduled commands (say, via cron(8)), their main purpose is to allow semi-interactive access to the system from the outside. A typical setup consists of a system account that is allowed to execute a pre-determined set of commands and a mechanism to authenticate and trigger remote invocations of said commands, such as retrieval or deposit of data files.

At Yahoo!, we usually perform these functions using a specific account included in our OS images with a custom restricted shell. This shell only allows a select set of commands (most notably the use of rsync(1), scp(1), and tar(1)) and is meant to let engineers safely transfer files between hosts in an automated fashion. The intention here is to avoid letting a headless user account run arbitrary commands that might be used to compromise a system.

The trust model for these kinds of accounts is somewhat complex: they are explicitly untrusted, but are simultaneously trusted to execute a very small set of commands only, because their use has been reviewed and determined to not pose a risk under the given circumstances (or, more precisely, the risk has been determined to be outweighed by the functionality gained by allowing this use).

The main problem with this approach is that it tries to impose a one-size-fits-all solution on all use cases. If a specific command needs to be added to the list of whitelisted executables, this requires careful review, as the command would then be made available to all users of this restricted shell. Thus, any and all use cases of the new command need to be considered (in contrast to the possibly very restrictive use case initially proposed). At a company the size of Yahoo!, this opens up a very wide field.

To illustrate the problem with this approach, consider the example of an often requested feature addition for this shell, namely, to allow execution of the ln(1) utility. Most specific use cases are non-controversial and ought to be allowed—however, ln(1) is also frequently used to set up an attack known as a "symlink race," which is based on a race condition when creating temporary files leading to

information exposure or corruption. For this reason, ln(1) executions cannot be approved as a generic command in this restricted shell.

Similarly, the existing feature in some shells to invoke a "restricted shell" (think `bash -r` or rksh(1)) has proven itself to be much too stringent for practical use. Many tasks that need to be run are impossible in such an environment; at the same time, many such implementations can trivially be circumvented (for example, by invoking an editor that allows you to invoke a new shell).

Headless Account with an Interactive Shell

Due to the shortcomings of the restricted shell, and at times also simply due to ignorance or laziness, some people set up headless users with a fully interactive shell (for example, /bin/bash).

The concern here is that this effectively opens up a regular user account for use by many people and other automated systems. The more people have access to the login credentials (i.e., the ssh keys used to authorize as the headless account), the more likely it is that these credentials might be compromised or abused, be that by way of accidental exposure (granting read permissions to the private ssh key to members outside of your team) or even maliciously. This, in turn, may lead to unauthorized access to a host and any of the data stored on the host, enabling a possible attacker not only to access specific data, but also to mount additional privilege escalation attacks from inside the host.

With an interactive login shell, these accounts fall into a bizarre state of simultaneously being completely trusted (effectively, by virtue of being able to run any given command) and explicitly not being trusted at all (implicitly, by being a headless user account; explicitly, by policy).

A Signature Verifying Shell

To overcome the above-mentioned issues, a new solution is needed. Engineers should be given the ability to define a wide range of commands to be run headlessly, but at the same time it must be ensured that they cannot cause problems by being invoked in unintended ways. For example, the headless account should be able to run `ln -s <dated-dir> <dir>` but not be allowed to `ln -s /etc/passwd /tmp/4jc5ba`, for example.

Once we change the goal from trying to determine a fixed list of commands that are always safe to execute—a difficult task, given the intentional flexibility of the UNIX operating system family—and shift our focus to the underlying trust model, it quickly becomes clear what is needed: a shell that verifies that the commands it is about to execute come from a trusted user, but allowing such users to run any command they choose. That is, we are not trying to protect the account from authorized users running "bad" commands, but, rather, access by unauthorized users to any commands.

To implement this new paradigm, we need a way to feed the shell a signature of the code to be executed and for the shell to be able to verify validity and trustworthiness of the signature. The most obvious solution for many people might be to implement such signature verification using a PGP-based approach. However, even though an entirely suitable solution, the UNIX-based tools that implement a PGP-based public-key infrastructure may not be part of the basic OS images as installed on all of your servers. To ensure ease of adoption of the new tool, the prerequisites

need to be restricted to the bare minimum. That is, the tool needs to work without requiring installation of any additional packages.

Fortunately, there is another set of tools that can be used to accomplish the same goal and which is included in most stock open source images as a core component. The openssl(1) utility implements the de facto industry public-key cryptography standard (PKCS) #7 secure message standard via its smime(1) utility, which allows, among other things, for signature creation and verification of secure/multipurpose Internet mail extension (S/MIME) messages as defined in RFC 5751. The signing of such a message simultaneously provides message integrity (the message was not modified in transport and is in fact what was sent) and authentication (the origin of the message is confirmed).

Even though S/MIME is, as the name suggests, mainly used in the context of email, a "message" can of course be anything—a shell script, for example. That means that you can authenticate and verify a given shell script so long as you have the right certificates installed on the host in question:

```
openssl smime -verify -inform pem -CAfile \
      /etc/sigsh.pem <input
```

The certificate file found in /etc/sigsh.pem contains the public-key certificates of all users who are authorized to sign commands for execution by this shell. A certificate is generated via a command like the following:

```
openssl req -x509 -nodes -days <expiration> \
      -newkey rsa:2048 -batch \
      -keyout <keyfile> -out <certfile>
```

It is worth noting that this certificate creation is a one-time step to be issued by the user in question and that /etc/sigsh.pem may contain any number of certificates. That is, multiple people can simultaneously be allowed to sign scripts for execution, eliminating the possibility of a single point of failure.

Once a certificate has been created and the public component has been installed on the desired hosts, a script can be signed for execution as follows:

```
openssl smime -sign -nodetach \
      -signer <public-cert> \
      -inkey <private-key> -in <script> \
      -outform pem
```

The result is a signed S/MIME message generated on stdout containing the given script. This can be either directly piped into an ssh(1) connection to the given host or simply stored in a separate file. Once signed, anybody can invoke the commands so long as they have access to the signed script. This is of particular importance, as it is desirable to limit the number of people able to sign scripts for remote execution by a headless account, but simultaneously to be able to let a larger number of engineers, or even other systems, run the commands headlessly without opening the door to let them run any other command.

Putting it all together, the following becomes a pipeline illustrating script signing, verification, and execution:

```
# script signing
openssl smime -sign -nodetach \
      -signer <public-cert> \
      -inkey <private-key> -in <script> \
```

```
      -outform pem   |  \
# script verification
openssl smime -verify -inform pem -CAfile \
      /etc/sigsh.pem  |  \
# script execution
/bin/sh -s
```

As a standard UNIX command pipeline, it is of course possible to redirect the output of any one of these steps into a file or to insert additional commands in between. For example, it would make sense to let a trusted engineer review and then sign the contents of the file script.sh into the file script.pem and then let another system execute this script on another host via:

```
cat script.pem | ssh headless@remote-host   \
       "openssl smime -verify -inform pem  \
       -CAfile /etc/sigsh.pem | /bin/sh -s"
```

This simple pipeline is the only thing the account on the remote host needs to be able to execute, yet using this construct it is possible to let it run any kind of command. In fact, this short openssl(1) command piping into a regular shell is the basis for the new tool we developed, sigsh(1). With some syntactic sugar, some assurance of meaningful exit codes, commentary, and the like, we managed to grow this simple pipeline to over 140 lines, but at its core sigsh(1) easily fits into a twitter message [3].

Threats Not Protected Against

Within the context of trust relationships between systems and users, headless users and commands executed, it is important to note that, while sigsh(1) provides assurance that the commands fed into it were the ones a trusted authority provided, there are a number of important caveats:

- sigsh(1) reads the list of certificates to trust from /etc/sigsh.pem, i.e., the local file system.
- /etc/sigsh.pem may contain multiple certificates.
- Certificates may expire.
- Host administrators have control over the certificates.
- sigsh(1) does not verify that the commands it executes are themselves trustworthy.

Looking at these items, a connection between the flexibility this program provides and possible issues can easily be seen. At the same time, it is worth remembering what we wanted—use of the headless user to gain unauthorized access to a host, that is, privilege escalation. We also made some assumptions, either explicitly or implicitly.

It is entirely true that a user with permissions to write to /etc/sigsh.pem can update that file to add their own certificate. However, an attacker capable of doing that already has, by definition, gained superuser privileges, and is able to install any other back door or wreak havoc in a myriad of other ways.

Multiple certificates, certificate expiration, and control over the certificates by host owners all provide precisely the desired flexibility: the respective drawbacks (multiple authorities, users forgetting to renew their certificates, the operational overhead involved in getting new certificates installed, engineers possibly removing a central authority's certificate, etc.) are all inherent in the design but are offset by exactly that flexibility, as required within the given threat model.

The last item listed above, however, deserves additional attention: it is worth stressing that protection against accidental execution of compromised binaries is not a goal. That is, the system does in fact assume that any of the commands fed into the shell after signature verification is indeed safe to execute.

Related Work

Our investigation of the initial problem statement led us to a number of interesting related projects. Within the UNIX world, there is a lot of focus on technologies that prevent accidental execution of compromised binaries, of detection of tampering with the system, and the like.

NetBSD, for example, has developed a file integrity subsystem named Veriexec [4]. This system allows you to permit execution of individual commands or command-interpreters only if they match a given signature. While this sounds similar to what sigsh(1) implements, it addresses a rather different threat model: here, the system does not verify that a sequence of commands was approved by a trustworthy entity to be executed but, rather, that a command, when it is to be executed, is in fact unmodified.

Other operating systems have similarly focused on this problem of a "trusted path." OpenBSD's Stephanie project initially developed a series of patches implementing "Trusted Path Execution" [5], which focuses on assurance of ownership and possible modifications of executables prior to invocation. A Linux Security Module was based on this work, similarly focusing on the integrity of the executables. While highly desirable functionality, it does not relate directly to the problems sigsh(1) was developed to address.

Early on in the conceptual development phase of sigsh(1), one system we encountered did, however, appear to exhibit all desired features: a shell that allows the administrator to set an "execution policy" specifying under what circumstances scripts may be executed, including settings that require a valid signature of said scripts. This tool had but one drawback: it only runs on Microsoft Windows.

The "Windows PowerShell" [6] is, much like a regular UNIX shell, both a command interpreter and a scripting language, which implements the concept of an "execution policy" that allows you to specify, for example, that any and all scripts executing using the PowerShell must be accompanied by a valid signature verified prior to execution.

It would be highly desirable to integrate the concept of multiple execution policies into our simple sigsh(1) implementation; at the same time, it would be nice if it were possible to allow signatures from a given certificate to work only for a subset of commands.

Of Signatures, Revocation of Privileges, and Audit Trails

One of the key points of the certificate-based solution is that certificates have an expiration date. That is, any given signing party may only be able to sign scripts for execution for a limited amount of time—this ensures regular audits of the list of certificates contained in /etc/sigsh.pem.

Similarly, certificates can be revoked. That is, if one of the people with the authority to sign scripts leaves the company, there is no need to wait for the certificate to expire. Instead, the ability to simply revoke the certificate and thus disable it without any changes occurring on the client hosts would be desirable. This, however,

would require sigsh(1) to check a certificate revocation list (CRL) and thus would introduce additional complexity (and network communications at runtime). It was decided that letting the hosts' configuration management system handle control of the contents of /etc/sigsh.pem was sufficient: removal of a trusted cert prior to its expiration thus becomes trivial.

Finally, sigsh(1) itself does not currently implement any sort of audit trail. While certificates added to /etc/sigsh.pem can be tracked via the configuration management system's changelog, input to the shell is executed without logging if it can be verified against the list of certificates found on the host. It would be desirable to have the shell log the commands executed, the identity of the signing party, and any errors or repeated signature mismatches. Future versions will likely include this ability, allowing you to discover and react to (intentional or accidental) misuse of the tool.

Conclusion

sigsh(1), Yahoo!'s simple signature verifying command interpreter, allows you to use headless accounts with arbitrary yet trusted input scripts by checking them against public-key certificates prior to execution. This removes a number of hurdles in the setup of complex interconnected systems that require asynchronous event triggering or data collection via non-trivial scripts and commands, and it improves overall system security.

The tool, implemented in only a few lines of code and using the universally available openssl(1) tool for all heavy lifting, puts the power of self-administration into the engineer's hands and eliminates cumbersome and overly restrictive solutions that are frequently circumvented. Deployed on over a quarter-million hosts, sigsh(1) has a proven track record across all divisions of Yahoo!.

In February of 2011, Yahoo! open sourced sigsh(1): it is available for use by anybody under a BSD-style license from github.com. Future enhancements may include more fine-grained control over who may sign what kinds of scripts or what kinds of signatures are required under what circumstances. Integration with a host-based file integrity check would then complete the goal of having assurance that commands executed by headless users are..."safe."

References

[1] sigsh—a signature verifying command interpreter: http://www.netmeister.org/apps/sigsh/.

[2] Don Norman, "When Security Gets in the Way": http://jnd.org/dn.mss/when_security_gets_in_the_way.html.

[3] https://twitter.com/#!/jschauma/status/35490334251294720.

[4] The NetBSD Veriexec subsystem: http://www.netbsd.org/docs/guide/en/chap-veriexec.html.

[5] Niki A. Rahimi, "Trusted Path Execution for the Linux 2.6 Kernel as a Linux Security Module," in *Proceedings of the 2004 USENIX Annual Technical Conference:* http://www.usenix.org/events/usenix04/tech/freenix/full_papers/rahimi/rahimi_html/index.html.

[6] Microsoft, "Windows PowerShell": http://technet.microsoft.com/en-us/library/bb978526.aspx.