Global Information Assurance Certification Paper

GIAC Security Essentials Certification Practical
Jason A. Richards
Version 1.4b
July 28, 2003

Auditing and Ensuring the Security of Homegrown Scripts

ABSTRACT

One of the greatest instruments available in a system administrator's tool belt is the ability to write scripts in order to automate routine and even not-so-routine tasks. Since becoming a system administrator myself, I have written hundreds of scripts, some that are simply a conglomeration of commands that I needed to execute I few times in a row, and then never again. Other times and more often, those scripts are written and stored for using over and over again, saving myself countless hours of work. While scripts can be very useful to a system administrator, they can also be and often are a huge security hole due to the speed with which they are written and the need to "get it done quick," rather than to "get it done right."

In this paper I present a methodology for auditing your network of systems for these scripts to identify whether they reveal any security concerns, how to proceed if they do, and I reveal common security problems with various scripting languages. This paper is written not only for the security conscious administrator, but also for managers and others that are less technical, but still need to be aware of these situations.

DEFINITION OF SCRIPTS

A fairly good definition of a script, taken from searchVB.com is "In computer programming, a script is a program or sequence of instructions that is interpreted or carried out by another program rather than by the computer processor (as a compiled program is)." In the context of this paper, a script is a collection of commands executed in a specific order that provides an expected output using a widely available interpreter such as perl, expect, bash, python, etc. This definition is intentionally wide ranging because the many tools that can be used to write and execute a script are quite different and I'm attempting to encompass them all here.

By this definition then, any user's .profile is a script, many operating system commands are scripts, everything written in perl is a script, etc. Again, in the context of this paper, we are not usually concerned with operating system provided scripts, although this area should not be overlooked. We're only concerned with scripts that have been written in-house by current and former employees that know passwords, trade secrets, system configurations, application vulnerabilities, backdoors, etc. It's the knowledge of these that lend

to gaping security holes in homegrown scripts. System administrators are also often asked to write web-based scripts that can be used by others in an organization in order to provide access to information without providing an actual system login. There are additional security implications of these types of scripts.

WHY SCRIPTS CAN BE UNSAFE

As mentioned in the previous paragraph, employees and contractors are trusted with sensitive information about the company they work for. This information commonly finds itself into scripts sometimes on a temporary basis and other times on a permanent basis. Examples of sensitive information commonly found in scripts:

- Sensitive passwords
- Key network infrastructure details
- Hostnames, supported protocols and port numbers
- Usernames

Certainly everyone can appreciate the need to keep passwords secret, but how about the other 3 examples given? Does your organization cover up posted network infrastructure diagrams during tours? Or better stated, do you provide copies of your network infrastructure to your direct competitors? Is your name server setup to allow anyone to pull an entire zone transfer of your domain(s), or are they only allowed to query for specific hosts? Do you make public a list of any or all usernames available on a system? These questions are important because they affect the confidentiality of your organization. Providing any other this information explicitly or implicitly (via insecure scripts) completes a large part of an attacker's reconnaissance mission.

At this point you may think that beyond potential confidential information revealed inside, a script is safe enough otherwise. This is simply not the case. Much like human beings, scripts too can be manipulated and tricked into doing things that we don't intend for them to do.

SECURING THE INSECURE

Now that we have a general understanding of how scripts, although helpful in many regards, can be a nuisance, we can discuss the steps necessary to eliminate their insecurities.

First and foremost, although quite basic, we need to cover the topic of file permissions. Anyone that is writing scripts on a system need to be familiar with and understand the implications of the file permissions they give to their scripts. Ideally, administrators are manually assigning permissions to their scripts as part of their process of developing it. Failure to specifically do so doesn't necessarily have to be a great concern however, if that person has set his/her umask. Umask is used to define default permissions for newly created files just for these

instances to ensure that scripts (and other files) can not be read/modify/deleted by anyone. Common umasks are 022 and 027. The former example sets all newly created files' permissions to 755. For the security-conscious I recommend the latter, which sets those permissions to 750. If you are unfamiliar with these concepts the Computer Science Department at Southern Illinois University offers a nice introduction at: http://www.cs.siu.edu/computing/unix/permissions.html

Knowing who the intended users are for the scripts that you are auditing/writing helps you to understand what file permissions you need to set on the script. For instance, if you are writing a log parsing script to be used by your technical support staff, you'll probably need to make the script readable and executable by the world, unless they are part of the group that the script is owned by. That brings us to a similar topic, file ownership. We already know that umask controls the permissions of newly created files, but what is the control mechanism for a new files owner and group owner? These are set to the owner and primary group of the user that creates the script (and other files). So if I create a script that I intend for technical support to use, I either have to change the group ownership of it to their group (assuming they are all part of a single group), or I have to modify the permissions so that the world can read and execute it. With that notion in mind, I should first sanitize the script and remove any critical information from it that I don't intend for them to see. Being aware of file permissions, then, helps prevent us from creating scripts with 777 permissions that contain confidential information.

Ensuring that our scripts are only readable / executable by those we intend is a good first step in this process. The next step is to ensure that those same people (or anyone that gains access to their accounts) can't use a script in a manner in which we didn't intend. There are a few ways in which a script can be tricked into doing something that we didn't intend. These are credited to David Totsch and taken from his article in *Enterprise Solutions* entitled "So, You Think Your Shell Scripts are Secure" available online at:
http://www.interex.org/pubcontent/enterprise/jul00/16uxsys.html

1. Modifying in-process temporary files
2. Exploiting the use of environment variables
3. Planting a specially formulated file for the script to operate on

Scripts are often used to create, modify and delete a variety of files usually requiring that we store some portion of the work in an external file while the script performs other functions until it needs that data again. The length of time that this temporary file lays in wait can vary from less than a second to countless minutes, all depending on what else the script needs to do. The problem with this is that if the script doesn't take appropriate caution in protecting that temporary file, it could be modified or even deleted, causing the script to act in unexpected ways. With the right modification, an attacker could use this to their advantage. This problem is of great significance because the locations that we

store these temporary, in-process files are usually /tmp or /var/tmp, both of which are world-writable. This means that anyone on the system can delete the temporary file, and if they know how the structure of the file, they can replace it with a file of their own. We can protect our scripts from this (to a certain extent) by first creating a directory with secure permissions and then storing our in-process files in that secure directory. That way an attacker can't even see the files we have stored, let alone the contents in them.

All too often we don't account for the environment variables that people have set when they execute our scripts. As usual, most of the time this doesn't cause a problem, but some of the time it does. If our shell scripts rely on the PATH environment variable for the location of 'echo' (or any other normally used command), then we could be in bad shape if the script executes as root and the user who is executing the script has a malicious 'echo' early on in their PATH. Historically I've always gotten around this by hard-coding the full path to the 'echo' that I want to use. This can cause problems however, specifically when my script is on an NFS share and I use it from multiple servers that are not identical in installation and configuration. A better solution, then, is to sanitize these environment variables at the beginning of the script. For instance, you could set the path to be '/usr/bin:/usr/sbin'. Generally speaking, that's a pretty safe PATH. PATH isn't the only environment variable we need to be concerned about, however. IFS, Input Field Seperator, is of equal or even greater concern. By having a carefully set IFS, a user executing a script can malform the scripts data and have it execute a list of commands (such as usr local bin echo) instead of the full path to a specific command (such as /usr/local/bin/echo). This then relies on PATH to find the usr, local, bin and echo commands. If one of those is malicious, problems will occur.

Lastly, piping multiple commands together in a script can leave our scripts open to malicious interpretation. Often times we'll write a simple script that lists the files in a directory and executes some operation on those files, be it a word count, a disk usage summary, etc. Take for instance, the command:

ls -1 | sed "s/^/wc -l /" | sh

We could be in a world of hurt if the directory we execute this in has the following files:

A;X=`echo \\\057var\\\057tmp`
B;PATH=$X:$PATH
C;Trojan

First, sh will count the lines in A, then it will assign /var/tmp to the variable X. Second, it will count the lines in B and modify the PATH such that /var/tmp is at the beginning. Thirdly, it will count the lines in C and execute Trojan. Presumably, there is a Trojan file in /var/tmp that has malicious code that could

mail a copy of the passwd and shadow files, or it could simply start deleting the entire contents of the disk using rm. A better way to do this is to use a for loop:

```
for I in *
do
 wc -l $I
done
```

In this manner, even if those same files exist, the X and PATH variables are only valid inside the current iteration of the for loop that they are executed. So this eliminates the issue. Now that we have covered some specifics to shell scripts, let's take a look at some other very popular scripting languages, such as perl, python and expect.

Perl is arguably the most useful scripting language I've ever used in administering systems. With that being said, there are two specific security issues with it that we should be aware of and account for.

1. Use perl in taint mode
2. Use sysopen() instead of open() (or use perl 5.6 in order to use open())

Perl can be enabled in taint mode by specifying the –t command line option (so the first line in your perl script becomes: #!/path/to/your/perl –t). In taint mode, perl checks various function calls for code that could be used maliciously. Although it can't account for every scenario, it does a pretty good job. For instance, it checks to make sure that we've specifically defined our environment variables such as PATH. We need to define these in perl just like in shell scripts, as mentioned above. A list of the monitored functions has been made available by Jordan Dimov, from his article "Security Issues in Perl Scripts: Perl Taint Mode" available online at: http://www.developer.com/open/article.php/631331.

If you're used to using perl's 'open' function in the following manner, you will want to consider using sysopen() instead:

```
open F, $file or print "couldn't open $file: $!";
```

Joseph N. Hall explains, from his article "effective perl programming" in *;login:* available online at: http://www.usenix.org/publications/login/2000-4/features/perl.html that the reason for this is because $file be set to "cat /etc/passwd" and it would do just that. The sysopen() function, on the other hand, is only able to open files, it can't execute commands, so using it would be more appropriate:

```
sysopen F, $file, O_RDONLY or print "couldn't open $file: $!";
```

Alternatively, if you have perl 5.6 or newer available to you, you can continue to use the open() function as long as it is used in the three arugment manner as such:

```
open (F, "<", $file) or print "couldn't open $file: $!";
```

A METHODOLOGY FOR AUDITING YOUR ENVIRONMENT

Now that we have discussed some issues with writing scripts, we need to develop a methodology for taking this new found information and applying it to your environment. If your platform(s) are brand new, you are the first administrator to operate them, and today is your first day on the job, then you have no reason to audit your systems for home-grown scripts because there aren't any. However, no one meeting those criteria is likely to come across this paper, so this section may very well be the most important. I'll use my environment as the example for forming this methodology.

Along with several other system administrators, I am responsible for almost 50 separate Unix servers hosting applications such as mail, DNS, web, DHCP, cablemodem provisioning, etc. Most of these systems have been up and running for more than year and have had several administrators that have performed work on them via a command prompt. Undoubtedly there are countless scripts on each system that have been written on the fly to solve a problem, and also as a planned upgrade to the system to enhance its usefulness. We need to be able to identify, review and correct any security issues that these scripts present (or at least document them so that the risk and exposure can be calculated and presented to the team and management.) If you can get management, or at least your teams buy-in that this is an important, time-sensitive issue, then I suggest you break the work out into components that each person can undertake individually, because one person may not have the patience or the thoroughness to complete the job satisfactorily. So in my environment, I would like to split the fifty systems into five groups so that each team member, including myself have 10 systems to audit.

Once the groups of systems have been identified and I've been assigned my group, I should perform the following steps:

1. **The discovery process** – This is the process I will need to follow in order to locate all of the home-grown scripts on each system. The order in which I execute each of the subtasks to this process is irrelevant, as long as I'm thorough and complete each one fully.

    a. **Check all users' crontab** – I've found that the most common place that scripts are referenced in a system are in the crontabs. Fairly often we write scripts that need to run on an iterative basis at specific times,

and that's exactly what cron is for. There's no need to run *crontab -l* for every user on the system, you can simply check the crontab directory. The Solaris Security FAQ, written by Peter Galvin and available online at: http://www.itworld.com/Comp/2377/security-faq/ tells us that on Solaris this is located in /var/spool/cron/crontabs. Redhat's support resources for cron, available online at: http://www.redhat.com/support/resources/tips/cron/cron.html reveal that on Redhat systems, they are located in /var/spool/cron. It may be different on your system, but executing a find or just manually poking around should reveal its location. In looking through each crontab, note each entry listed and then check the actual file referenced to see if it is an operating system command such as rdate, or if it's a home-grown script. Then add this script to your list.

b. **Check the process list** – Executing *ps -ef* or *ps auxw* will give you a listing of the processes currently running on the system. This can be a good clue as to what home-grown scripts users are using that may or may not otherwise be easily determinable.

c. **Check users' home directories** – Although this can lend itself to providing you with a comprehensive list of scripts to check, you need to exercise caution in this matter and be sure you have explicit written permission to do so. Users are likely not going to appreciate you "snooping" around in their personal space, even if they have nothing to hide. If you do have permission, however, specifically check for ./bin and ./scripts subdirectories.

d. **Compare all files on a system to similar systems** – With regard to file systems local to a Unix machine, this step is generally a catch-all once you have completed the steps above. By executing a well-crafted find command, you can generate a listing of all files on the system and then compare it to a similar listing from another like machine to generate the differences. A "like" machine is a machine of the same OS installation, patch-level and preferably install date. Comparing a Solaris 9 system that was built 6 months ago to a RedHat 6.2 installation from 3 years ago is likely to be very unproductive whereas comparing two Solaris 8 machines that were built 2 summers ago, even though the provide completely different services is worthwhile. This task by no means is not easy, but it will catch all of the scripts you missed before (including those in /var/tmp, /usr/local/bin, etc.) A great tutorial on using find has been written by Computing Services – Web Unit of Athabasca University and is available online at: http://www.athabascau.ca/html/depts/compserv/webunit/HOWTO/find.htm

e. **Systematically cover shared file systems** – In any medium to large business, this step is likely to be the most challenging. Many companies find it useful to have a universal shared file system that extends across a large portion of, if not all of its systems. Since there is only one, you have nothing to compare it to, like in the step above. This task is best undertaken over time, and by as many administrators that you can involve as possible.

2. **The auditing process** – Now that you (and hopefully your coworkers) have developed a comprehensive list of the home-grown scripts in use in your organization, you can combine duplicate entries (such as a script that was written on one system and copied **without changes** to one or more other systems.) If your organization has been around for a while then you're likely to develop a lengthy list of scripts to be reviewed. Again, this process may best be undertaken by dividing out portions of the list to each available administrator. The meat of this process, then is to review each script that you've been assigned for not only the issues discussed above, but for other security issues that you'll need to research on your own. By no means has this paper covered all, or even a wide-range of security issues related to home-grown scripts.

3. **The implementation process** – After having reviewed each of the home-grown scripts from your list. You should have come up a replacement script, if you have the technical knowledge, or at least a documented list of the potential security implications. Regrouping with your fellow administrators, it now becomes time to cover any scripts that the individual administrator couldn't rewrite / correct on his or her own. Having done this, each script also needs to be reviewed to see whether the changes will impact the scripts currently in production. Your organization's change management policies will dictate whether you can just replace an insecure script with a more secure script. In my organization, for instance, there are many scripts that we use on a regular basis that simply provide information, are not used by any automated processes, and are expendable. Those criteria enable me to fix and replace the script at will without following our change management policies. On the other hand, we also have dozens of scripts that are used for providing customer-facing data, pulling billing reports, and are not expendable. The changes to these scripts need to be scrutinized, tested and retested to ensure that they don't impact the existing service when they are put into production.

CONCLUSION

Home-grown scripts are necessary and immeasurably advantageous to systems administrators. To do away with them is not only foolish but impossible. The key is to learn how to write them securely and to fix the scripts already written that are insecure. This process is an iterative process and will develop over the years

that we work in this field. System administrators need to be conscious of this fact and need to be resourceful in using the Internet to discover these security implications, to learn the fixes for the discovered implications, and to be aware of best practices used by others in the Internet community.

Auditing the existing environment is crucial to being prepared for attacks, both internal and external. If you can circumvent an attackers reconnaissance mission, you can likely already circumvented the attack. Therefore, securing your homegrown scripts is as crucial to your organization as implementing a firewall, running and regularly updating your virus software and monitoring your systems for unusual events.

REFERENCES

Computing Services – Web Unit, Athabasca University. "Some examples of using UNIX find command." March 30, 1999. http://www.athabascau.ca/html/depts/compserv/webunit/HOWTO/find.htm

Department of Computer Science, Southern Illinois University. "Unix Basics, File Permissions." 05/10/2000. http://www.cs.siu.edu/computing/unix/permissions.html

Dimov, Jordan. "Security Issues in Perl Scripts: Perl Taint Mode." Developer.com. February 7, 2001. http://www.developer.com/open/article.php/631331

Galvin, Peter. "The Solaris Security FAQ." Unix Insider. January 1, 2001. http://www.itworld.com/Comp/2377/security-faq/

Hall, Joseph. "effective perl programming." ;login. April, 2000. http://www.usenix.org/publications/login/2000-4/features/perl.html

Redhat, Inc. "Cron." 2003. http://www.redhat.com/support/resources/tips/cron/cron.html

searchVB.com. August 27, 2002. http://searchvb.techtarget.com/sDefinition/0,,sid8_gci212948,00.html

Totsch, David. "So, You Think Your Shell Scripts are Secure." Enterprise Solutions. July, 2000. http://www.interex.org/pubcontent/enterprise/jul00/16uxsys.html

Last Updated: July 21st, 2017

Upcoming Training

SANSFIRE 2017	Washington, DC	Jul 22, 2017 - Jul 29, 2017	Live Event
SANS Boston 2017	Boston, MA	Aug 07, 2017 - Aug 12, 2017	Live Event
SANS Prague 2017	Prague, Czech Republic	Aug 07, 2017 - Aug 12, 2017	Live Event
Community SANS Omaha SEC401*	Omaha, NE	Aug 14, 2017 - Aug 19, 2017	Community SANS
SANS Salt Lake City 2017	Salt Lake City, UT	Aug 14, 2017 - Aug 19, 2017	Live Event
SANS New York City 2017	New York City, NY	Aug 14, 2017 - Aug 19, 2017	Live Event
Community SANS Trenton SEC401	Trenton, NJ	Aug 21, 2017 - Aug 26, 2017	Community SANS
Virginia Beach 2017 - SEC401: Security Essentials Bootcamp Style	Virginia Beach, VA	Aug 21, 2017 - Aug 26, 2017	vLive
SANS Virginia Beach 2017	Virginia Beach, VA	Aug 21, 2017 - Sep 01, 2017	Live Event
SANS Chicago 2017	Chicago, IL	Aug 21, 2017 - Aug 26, 2017	Live Event
SANS Adelaide 2017	Adelaide, Australia	Aug 21, 2017 - Aug 26, 2017	Live Event
Community SANS Pasadena SEC401 @ NASA	Pasadena, CA	Aug 23, 2017 - Aug 30, 2017	Community SANS
Mentor Session - SEC401	Minneapolis, MN	Aug 29, 2017 - Oct 10, 2017	Mentor
SANS San Francisco Fall 2017	San Francisco, CA	Sep 05, 2017 - Sep 10, 2017	Live Event
SANS Tampa - Clearwater 2017	Clearwater, FL	Sep 05, 2017 - Sep 10, 2017	Live Event
Mentor Session - SEC401	Edmonton, AB	Sep 06, 2017 - Oct 18, 2017	Mentor
SANS Network Security 2017	Las Vegas, NV	Sep 10, 2017 - Sep 17, 2017	Live Event
Community SANS Albany SEC401	Albany, NY	Sep 11, 2017 - Sep 16, 2017	Community SANS
Mentor Session - SEC401	Ventura, CA	Sep 11, 2017 - Oct 12, 2017	Mentor
Community SANS Columbia SEC401	Columbia, MD	Sep 18, 2017 - Sep 23, 2017	Community SANS
Community SANS Dallas SEC401	Dallas, TX	Sep 18, 2017 - Sep 23, 2017	Community SANS
Rocky Mountain Fall 2017	Denver, CO	Sep 25, 2017 - Sep 30, 2017	Live Event
SANS London September 2017	London, United Kingdom	Sep 25, 2017 - Sep 30, 2017	Live Event
SANS Baltimore Fall 2017	Baltimore, MD	Sep 25, 2017 - Sep 30, 2017	Live Event
SANS Copenhagen 2017	Copenhagen, Denmark	Sep 25, 2017 - Sep 30, 2017	Live Event
Community SANS Boise SEC401	Boise, ID	Sep 25, 2017 - Sep 30, 2017	Community SANS
Baltimore Fall 2017 - SEC401: Security Essentials Bootcamp Style	Baltimore, MD	Sep 25, 2017 - Sep 30, 2017	vLive
Community SANS New York SEC401	New York, NY	Sep 25, 2017 - Sep 30, 2017	Community SANS
SANS DFIR Prague 2017	Prague, Czech Republic	Oct 02, 2017 - Oct 08, 2017	Live Event
Community SANS Charleston SEC401	Charleston, SC	Oct 02, 2017 - Oct 07, 2017	Community SANS
Community SANS Sacramento SEC401	Sacramento, CA	Oct 02, 2017 - Oct 07, 2017	Community SANS