

8

Script Safety and Security

As we have tried to demonstrate throughout this book, Perl scripts can save an enormous amount of time and frustration if you are faced with the task of maintaining or configuring a large number of Windows NT workstations. Once you have put the initial effort into setting up a protocol and infrastructure for your scripting, even the most complicated of tasks can be readily accomplished without a hitch. From time to time, however, it is almost inevitable that something will go wrong. Even if you are meticulous when creating your code and you test every line that you write, occasionally something will have an unanticipated side effect and cause a great deal of frustration. At this point you will ask yourself why you ever bothered with all this scripting nonsense! The answer, of course—as you will know deep down—is that the vast majority of the time, things do not go wrong; on balance, implementing pretty much anything with a well-designed script leads to a far smoother outcome than if you try to achieve the same effect manually. Unfortunately, we cannot give you a magic formula to ensure that your scripts never go wrong. What we can do, however, is give you a few guidelines for safe scripting and point out a few things that tend to cause problems. Ensuring as much as possible that scripts leave workstations in a stable, secure state is the subject of this chapter. We will begin with some general guidelines for scripting and suggest how to avoid some of the more common errors. In the second half of the chapter, we will address the issue of script security and how to prevent a malicious hacker from turning your own scripts against you.

Development and Deployment

It may be impossible to predict the exact effect your maintenance and configuration scripts will have under all conceivable circumstances, but it is normally very

easy to predict the most likely outcome in almost all circumstances.* After all, if this were not the case, there would be no point in scripting at all. If there is a trick to making sure your scripts work, it is to have a consistent development and deployment strategy; then at least, if things go wrong, you will know exactly what you have done (and therefore what needs to be undone).

Some Development Guidelines

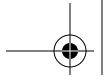
The most important tool you can possibly possess when you are writing administration scripts is patience. This may seem like quite a weird statement to make, given that the main purpose of scripting is to save time, but as we have said before and will no doubt say again, the savings comes from the fact that a script has to be written only once and emphatically *not* because it is quick to write in itself. It is so tempting to sit down, rush out a simple script, tweak it until it works, deploy it on your workstations, and get on with the next job. However, this is the classic way of prompting disaster. Just as you would never type `del *.* /q` at the command prompt of your main server without checking and double-checking that you are in the right place in the directory hierarchy, you should never deploy a Perl script on even a single workstation unless you are absolutely sure what it will do and what it will do it to.† Remember that a badly written script running with administrative or system privileges on a workstation can cause just as much damage, if not more. In short, a methodical, calm approach to scriptwriting is always worth the effort.

Next, we present a set of guidelines for script development that, in our opinion, greatly limit the potential for making a serious mistake. The formula we propose is quite strict, and it is inevitable that you will deviate from it from time to time. Nevertheless, it provides an idealized framework that you should keep in the back of your mind whenever you are writing scripts:

1. Define your project. Before you write a single line of Perl, you should have a clear idea of exactly what it is that you want the script to achieve. It may seem obvious that you can't write a script to do something unless you know what you want it to do, but it is a point that is often overlooked. If your project is at

* Occasionally, the most obscure anomaly in software or even hardware design will make it totally impossible to predict the outcome of even a straightforward(ish) computation. Remember the Pentium FDIV bug?

† You can, of course, combine the two types of sin by writing a Perl script that accidentally invokes `del *.* /q` on completely the wrong part of the directory hierarchy. One of the authors of this book has suffered at the hands of just such a script written by the other. The script was meant to delete stuff selectively based on a regular expression search. Despite assurances from the one who wrote the script that it worked perfectly, the one who ran it managed to get the whole of his home directory wiped. If you can guess which author is which, we'll send you a free copy of the script. You can even have distribution rights!



all complicated, it is invariably worth writing down a specification of its functionality.

2. Work out how to implement the task. Unless you are a real expert, it is very unlikely that once you have defined your task you can just sit down and write the code. The first question to ask is whether the task is possible. Again, this may seem obvious, but it is amazing how much time you can waste trying to solve an insoluble problem or at least achieving an impractical solution! If your task involves interaction with the operating system, how is this best achieved? Do you need command-line utilities or Perl modules? In the former case, work out how you are going to extract any feedback you require from the utility (regular expressions? return values?). In the latter case, make sure you understand how the module works; if the documentation is sparse (as is sometimes the case), write a small script just to try out the module and make sure you understand it.
3. Write the code. If you have carried out the first two steps thoroughly, this is the easy bit! If it contains complicated regular expressions or anything out of the ordinary, it is invariably worth trying the relevant sections in an isolated script of their own before incorporating them into the main script. One of the huge advantages of Perl over a traditional compiled language is that it is very easy to try out isolated code fragments, so make use of this feature; it is invariably quicker and safer to ensure something works when you write it than to try and debug later. At this stage of script development, the very act of writing may well give you new ideas, or maybe something you read in a module's documentation gives you an idea for how you could add that extra bit of functionality. However, unless a change is very localized or unless there is an *extremely* good reason to deviate, always stick to your original game plans. If you have a good idea, jot it down and use it next time; if it's that revolutionary, maybe begin writing your script again to incorporate the new feature. But by changing your plans on the fly, you are asking for trouble and unforeseen implications.*
4. Test your code. Just because something *should* work, there is no guarantee that it *will* work. The degree to which you need to test your script clearly depends on the implications of a malfunction. What happens if it doesn't do what it should? Does it do anything that can be considered dangerous (i.e., the `del *.* /q` scenario)? If your scripts construct any potentially lethal commands with a series of variables and regular expressions, a very useful debugging trick is to enclose the command line within a `print` statement; you can

* We should make it clear that we are certainly *not* against fresh ideas; we actively encourage them. However, these ideas should be incorporated into a well-considered strategy, not just implemented in an ad hoc fashion.

then see exactly what it will do without running the risk of executing it with mistakes. Another thing you should do as a matter of course is run your script at least once with Perl's `-w` flag (e.g., `C:\>perl -w myscript.pl`). This will warn you of any ambiguities or errors in your code that the compiler would normally ignore.* If the script has conditional execution blocks, test all possible conditions; don't assume that if one works, they all will. Finally, test your script on a workstation that can be trashed without repercussions!

5. Once you are confident that your script works, deploy it (see the next section).

A further point to bear in mind is that you should always make your scripts readable. It can often be tempting to write obscure, clever bits of code that look impressive on the page, but this is generally not a good thing to do. Not only is it a pain when you (or someone else) has to modify the script at a later date, but it also makes it very difficult to debug. Ensuring that your code is transparent and easy to comprehend is a challenge that you should *always* take up. If you do use weird code, employ a particularly complicated regular expression, or exploit an idiosyncrasy of a module that you are using, supply copious comments (`#`). It is always worth giving a colleague your script to read and asking him to tell you what it does; if he can't, it is probably not clear enough.

Methods of Deployment

Once you have written and tested your script and are confident that it does what it should, it is ready to be deployed on your workstations. If it is designed as a tool to be used manually by administrators on an ad hoc basis, deployment may simply involve placing it on a server drive or even a floppy disk; it can then be run from the command line when required. If, however, the script is meant to carry out maintenance on a fleet of workstations (either as a one-off exercise or on a regular basis), it will need to be installed on all relevant workstations. Depending on how you have set up your workstation environment, this may consist of dropping the script into a server drive (to be run by a stub—see Chapter 3, *Remote Script Management*) or installing it directly, using one of the methods described in Chapter 2, *Running a Script Without User Intervention*. Whichever method you choose, there are a few helpful things to bear in mind:

- Even a script that has been written faultlessly could potentially cause havoc if used incorrectly. To return to our favorite example, a script designed to wipe the contents of a directory tree could have disastrous consequences if run from the wrong location, even if the script itself works beautifully.

* Running Perl with this flag actually produces a lot of very useful diagnostics. For full details, see the *perldiag* page in the online documentation supplied with the *ActiveState* distribution.

- If your script makes assumptions about how a workstation is configured, ensure either that the machines on which you run it conform to these assumptions or that failure to conform does not cause a problem. Even if the consequences of a failure are not disastrous, there is no point in running a script that cannot be guaranteed to carry out its intended task.
- Ensure that a run-once configuration script that has been installed directly to a group of workstations (i.e., run automatically but not by a stub) uninstalls itself properly once it has carried out its task and leaves the workstation in a usable state. For example, a typical error might be for a script that runs using the `AutoAdminLogon` registry feature (see Chapter 2) to fail to switch off automatic logon. Such a mistake will not only make a workstation unusable, but could also be rather insecure.

We hope that by now the message is clear: methodical thought and careful planning make certain that scripting is an administrator's savior and not a troublesome liability!



Security

At several points in the book we have made reference to the security implications of various aspects of scripting. As yet, however, we have not discussed script security as a subject in its own right, despite the fact that it is clearly an extremely important issue. For the remainder of this chapter, we highlight some important security issues that arise when you use scripts to carry out administrative tasks and suggest how to avoid falling into pitfalls.



User Accounts for Scripting

Scripts cannot just run by themselves; they have to be run by somebody (or by something). Just like any other program, a running script adopts the security context of that somebody (or something) that ran it. The implication of this is that the extent to which even a disastrous script can wreak havoc is strictly limited by the extent to which its owner has control over a workstation.* This is an extremely comforting thought: even if you have made a terrible mistake while writing a script (for example, creating a `del *.* /q` scenario), nothing serious can possibly go wrong provided you execute the script in the security context of a user who lacks dangerous privileges (like full control over the file system!). The only slight snag is that if you restrict your scripts in this way, they will probably not be able to function at all. For example, a script whose purpose is to delete the contents of temporary directories cannot carry out its task unless it has delete permissions on the file system (or at least on the relevant part of the file system).

* We refer here to the owner of the running process; this is not necessarily the same as the owner of the file.



Techniques we have shown you so far in this book involve editing the registry, reading event logs, writing to the file system, and all sorts of things that require administrative permissions in order to do their job. We have thus far avoided the issue by assuming an Administrator or LocalSystem security context (the default context for anything running as a service). In the real world, however, it may well be very foolhardy to allow all your scripts to run in such powerful contexts. In anything but the most trivial situations, it is worth thinking very carefully about the security context in which scripts should run; this normally involves setting up a special account with permissions optimized for scripting.



In an ideal world, every script you ever use will run in a security context designed especially for it; the aim would be to allow it to carry out all required tasks but nothing more. Realistically, however, this is not possible or even that desirable: not only would it require a huge effort to create special user accounts every time you created a script or modified an existing one, but the complications would greatly increase the chances of something going wrong. A far better solution in most situations is to have a single account set up for scripting; this account would have all the permissions that tend to be required for scripting but no more than required. In a particularly sensitive environment, you could even have a handful of special accounts, each associated with a stub script and a set of permissions; whenever you add a new script, you add it to the appropriate stub directory depending on the amount of power the script needs. When making decisions about a script's security context, what is required is a balance between safety and functionality.

When creating an account for scripting, the issues are not necessarily localized on a workstation. An average script may require only permissions on a workstation, but some (notably stub scripts) also require read access to a server share; a script that carries out a reporting or archiving task may even require write access to a server share. Throwing caution to the wind, an obvious way to deal with this would be to create a domain account that has administrative permissions. As we've said before, however, the obvious solution is not always the best. If you create a domain account with administrative permissions, these will apply on *all* computers within the domain; that means not only to workstations but also to servers and even domain controllers. The potential for serious damage if a script running in such a powerful security context goes wrong—or if it is compromised by a malicious cracker—are horrendous.

Bearing in mind that many scripts need lots of permissions on the workstations on which they are running, what is the alternative? One feasible possibility is to create two types of scripting accounts, one on the server side and one on the workstation side. If both types of accounts share a username and password, a script running on a workstation will be able to connect to server shares transparently but

will not have administrative rights on these shares (or the servers that contain them). The advantages of this scenario are clear:

- The scope of control of a local user account is drastically limited, compared with an equivalent domain account. Such an account can be used only on machines for which specific authority has been granted.
- Local accounts can have different security levels. For instance, the fact that a workstation account with the same username and password as a server account may have full administration rights on that workstation is of no significance to the server.
- Only machines that need to participate in this scripting arrangement need have local scripting accounts. Any other machines in the domain remain blissfully unaware of this special account and are safe from any compromise.

In short, the balance between safety and freedom to carry out necessary functions for a script is best met by creating a separate local account on each workstation to be scripted. This account should have a flexible enough set of permissions to handle most scripting requirements. Equivalent server accounts should be set up with a highly restricted set of permissions, so that if workstation security is compromised, your entire network is not at risk.

Keeping the Script Safe

Creating special accounts is an essential requirement for safe scripting, but this is only one aspect of script security. It is equally important to ensure that nobody can tamper with script directories on servers or workstations, inadvertently (or deliberately) turning a safe script into a lethal one. In some cases, even allowing users to *read* a script constitutes a serious breach of security. A few reasons for this are outlined here:

- If a script can be read, it can talk. The most obvious example of “script talk” is when a script contains sensitive information. For example, it is sometimes necessary for a script to contain a username and password, say to connect to a network resource or configure a workstation for autologon. There is nothing wrong with having such information stored in a script per se—it is a good way of allowing certain scripts to have access to a server resource without the need to create a special user account—but such scripts should not be readable by *anyone* except an administrator, inasmuch as this is possible.*

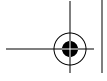
* If you are connecting to a server resource and sending plain-text passwords over the network, bear in mind the possibility that some cracker somewhere on your subnet might have a promiscuous network card.

- Username and password are not the only example of sensitive information to be found in a script. In some environments, there are also more subtle concerns to bear in mind. For instance, you wouldn't have thought there was anything worrying about a script that sends a `winnpopup` message to an administrator's machine. However, someone with malicious intent might be very grateful for such a script; it will tell her where to install a packet-sniffer to increase the likelihood of grabbing useful passwords.
- It is essential that a script cannot be modified by anyone except the administrator. A given user may have very few rights on a machine, but if he is able to modify a maintenance script that runs regularly with lots of rights, he is able to do anything he likes. A would-be miscreant could easily replace a script with one that adds him to the Administrator group. All he has to do is make the change and wait until the script runs.
- If you run scripts from a stub, it is not only modifications you need to look out for but also additions. If a user is able to add a new file to the script directory, it would be an easy task to drop a new, ready-to-do-damage script in place. Of course, if the stub connects to a server to update its scripts before running, the miscreant's code will be wiped, but it is not a good idea to rely on this. Anyway, if the server drive is compromised, a glaring Achilles' heel will be exposed!

We apologize if we seem to be painting a rather miserable picture, and we certainly do not want to seem alarmist, but security is always an issue to be taken seriously. The good news is that it is not very difficult to ensure that your scripts are free from prying eyes and malevolent users. The key to protecting yourself from a script-based attack is to ensure that the access control list for each script and for a stub-script directory grants access only to those people who need it. In most circumstances, this will be restricted to the account in whose security context scripts actually run, and the Administrator group. Further, the script account on a server should have only read permissions on the directory containing scripts for updating. This simple rule, in conjunction with awareness of the issues and a bit of common sense, should ensure that administration scripts do not compromise your network security at all.

Summary

As an administrator who makes heavy use of scripting to carry out maintenance and configuration tasks on Windows NT workstations, you can save yourself enormous amounts of time and frustration. Your workstations are also likely to have far better track records for consistency and reliability than those of your nonscripting colleagues; a well-written, sensibly deployed script will behave more predictably



Summary

than any human can! However, the power of scripting brings with it some potential pitfalls: an ill-conceived, buggy script running with administrative privileges can cause a catastrophe very quickly indeed. Further, careless deployment could lead to the sort of security hole on your network that will make you extremely unpopular with your boss. In this chapter we have discussed strategies that you can use to avoid falling into any serious pitfalls and to ensure that all of your scripts run safely and securely.

