Exploiting environment variables

G. Lettieri

3 October 2020

1 Introduction

Now let us play the role of an attacker. We assume that we have a normal account on a Unix system, and we want to *escalate* our privileges—possibly "become" root. By this we mean that we want to be able to run programs of our choice in processes with an effective uid of 0. Our ultimate goal is to get a "root prompt", the "`# `" prompt that the Bourne shell prints out when running as root.

Since we are a normal user and our uid is greater than zero, the `login` program will call `setuid(uid)` when we log into the system. This uid will be inherited by our shell. Assuming that we cannot tamper with the system up to this point, and that the kernel is working correctly, we now have only two ways to run programs with an uid other than our own:

1. somehow make processes that have inherited a different uid run what we want;
2. somehow make available set-uid programs run what we want.

Both of these methods require that the legitimate owners and/or creators of such processes (case 1) or programs (case 2) make mistakes. In a properly configured system, we shouldn't be able to do any harm: processes owned by other users will run programs that we can't control, and set-uid programs will do what they are supposed to do without any interference from us. Unfortunately (for the victims, but fortunately for the attackers), mistakes are very easy to make, especially in a highly configurable system that was developed in an environment very different from our own: a small circle of people where everyone knows everyone else.

2 Exploiting `PATH`

Suppose we want user u to run a program of our own choice, let us call it p. When user u types a command like c at her shell prompt, the shell will search a set of directories, depending on the contents of u's `PATH` variable, for a file called c. If we can control either u's `PATH` variable, or the contents of a directory that comes before the one containing c in the search list, we can make u unvoluntarily execute our own p when she wants to execute c: just rename p to c and copy it into the controlled directory. The victim's shell will find our fake c before the legitimate one and execute it, using u's credentials of course.

For this *attack vector* to work, however, we need u to make some mistake. We normally have no way of influencing the contents of u's `PATH` variable, whose value is set in the chain of processes that leads from `init` to her shell, which we assume is out of our reach. Moreover, if u's `PATH` variable only mentions directories that we cannot write to, such as `/bin`, `/usr/bin`, and so on, our attack options are zero.

However, if we boot our PDP-11, log in as root in our freshly installed Unix V7 system, and type `echo $PATH`, our Teletype ASR 33 will print out the following:

```
:/bin/:/usr/bin
```

By default, the root user has an empty path in her `PATH` (did you notice the first colon?) What's more, this empty path comes before the other directories. This means that whenever the system administrator types a command at her prompt, her shell will look for a matching executable file in her current directory before looking anywhere else[1]

If root has not changed the default, this is the mistake we need. Now all we have to do is put our *attack payload* (the program that we want root to run) into the directories where we have write access (hour home, or `/tmp`), give it the name of some common utility (`ls`, `find`, `cat` or whatever), and wait for root to `cd` there and execute the payload for us unnoticed.

A possible payload is the following:

```
cp /bin/sh /home/attacker/hello.c~
chmod u+s /home/attacker/hello.c~
```

We make a copy of the shell with a filename that looks like something else (like the swap file of some editor) and set the set-uid flag on it. Remember that these commands will be run by root. Therefore, `hello.c~` is now a shell that gives root access to anyone who runs it.

[1] This was done by design. In the earliest Unix implementations, Thompson's shell searched for executables first in the current directory, and then in the `/bin` directory: the idea was that users where expected to be programmers, and would want to run their own programs, created in their own directory; `/bin` was searched *after* the user's directory, to avoid accidentally hiding user programs with omonimous system programs. In V3, the shell also tried `/usr/bin` after `/bin`: this new directory was introduced when the available space in `/bin` was exhausted (in fact, the V3 manual calls the `/usr/bin` files "overflow" programs—V4 hides this bit of history and tries to repurpose `/usr/bin` as a place for "lesser used" programs, presumably because it was searched for last). When environment variables were introduced in V7, the Bourne shell generalized the search for executables with the `PATH` variable, following the example of the PWB shell; the default value was chosen to reproduce the old behavior.

Exercises

2.1. Try the above attack in the *badpath* challenge. Be careful: if something doesn't work as expected, root may find out what you are up to.

The attack is a bit risky, since an "`ls /tmp`" from a safe directory, an "`echo *`" from `/tmp`, an explicit call to "`/bin/ls`", and so on, will easily make the administrator suspicious, and an `ls -l /tmp/ls` will also reveal our name as the owner of the script. However, it can be very effective as a step in a longer privilege escalation chain—we may have stolen the account of another regular user (e.g., by guessing their password) and created the script using that intermediate victim's credentials.

3 Exploiting setuid programs

Now let us try to exploit the second possible attack vector: vulnerable set-uid programs. These programs must be written very carefully and, as a rule, they should not trust anything coming from the outside: command line arguments, environment variables, open files, directories writable by untrusted users—the list is unfortunately very long.

Set-uid programs are the favorite targets of attackers with login access to a Unix system (also known as *local* attackers), and we will examine their possible vulnerabilities in several lectures. Here we examine some vulnerabilities that are mostly of historical interest. However, they are helpful in introducing the topic.

Suppose a novice programmer writes a set-uid program that uses the `system()` library function, such as:

```
#include <stdlib.h>
int main()
{
    // stuff
    system("grep something somefile");
    // other stuff
}
```

The programmer needed a functionality similar to that provided by the `grep` utility, so she decided to reuse `grep` itself. The problem is not `grep`: it could have been anything. The problem is that the `system(cmd)` works by `fork()`ing a process and making it run

```
    /bin/sh -c cmd
```

The shell will parse `cmd` according to its usual rules, including using `PATH` to look for `grep`. Now, however, the attacker has a major advantage over the scenario in section 2: the shell created by `system()` will inherit the environment of the setuid program; unless the programmer explicitly cleans it up, the setuid program's environment will be the one inherited from the parent process, i.e., *the attacker's shell.*

Exercises

3.1. Exploit the above idea to obtain a root shell in the *bad0suid* challenge.

3.2. The *bad1suid* challenge contains a setuid binary with a different kind of vulnerability. Try to obain a root shell from it too.

Notice how vulnerabilities in set-uid programs are much better, from an attacker's point of view, than vulnerabilities like the one we examined in Section 2. In the "dot in `PATH`" vulnerability, there are many things that are not under the control of the attacker, who just has to wait for them to happen by accident: root (or another user) must have put the dot in her `PATH`, she has to `cd` into the directory where the attacker has planted the trap, she has to execute the fake command. Errors in the attack payload can also render the attack ineffective, and the attacker must wait for the entire sequence of events to occur again, which also increases the chances of getting caught. Vulnerable set-uid programs, on the other hand, are an attacker's dream: she can control essentially the entire execution environment, and she can run them at will.

3.1 Exploiting the IFS variable

Now suppose that the inexperienced programmer tries to patch the vulnerability in the following way:

```
#include <stdlib.h>
int main()
{
    // stuff
    system("/bin/grep something somefile");
    // other stuff
}
```

Since `/bin/grep` is a path, the shell will not use the `PATH` variable. Also, since the path is absolute and only traverses directories writeable only by root, it must lead to the real `grep` utility.

Let's put on our attacker hat again. While thinking of ways to exploit the new program, we type the following into our V7 shell:

```
  IFS=,
  ls,-l
```

Perhaps surprisingly, our teletype starts printing the long listing of the current directory. What we have done is to change the value of the `IFS` variable, which contains the characters that the shell uses as field separators (`IFS` stands for Internal Fields Separator). After parsing the command line into words and operators, the shell examines each word for possible expansions (e.g., processing `$`*variable* expressions) followed by *field-splitting*. The latter processing uses `IFS` to split words into fields, which then become the actual arguments used to execute a command. The default value of `IFS` is ⟨space⟩⟨tab⟩⟨newline⟩, but now we have changed it to a comma. This splits "`ls,-l`" into "`ls`" and "`-l`", resulting in a normal call to the `ls` program with the `-l` option. The 9-fields revision of the elementary shell of Chapter **??** supports `IFS`. The processing is implemented in a new function `fieldsplit()` that is called by `expandall()` on every normal word, after `expandword()` has finished.

Exercises

3.1. Abuse `IFS` to get a root shell from the vulnerable seduid program in the *bad2suid* challenge. This challenge uses a shell (`bad2sh`) that uses `IFS` the way the Bourne shell did: all the characters in `IFS` are equivalent to whitespace. You can see the code in the 9-fields revision of the elementary shell.

3.2. The `bad3sh` used in the *bad3suid* challenge implements `IFS` processing in a different way, as mandated by POSIX. In particular, point 3 of Section 2.6.5 of the standard says:

> [. . .] The term "`IFS` white space" is used to mean any sequence (zero or more instances) of white-space characters that are in the `IFS` value (for example, if `IFS` contains ⟨space⟩⟨comma⟩⟨tab⟩, any sequence of ⟨space⟩ and ⟨tab⟩ characters is considered `IFS` white space).
>
> a. `IFS` white space shall be ignored at the beginning and end of the input.
>
> b. Each occurrence in the input of an `IFS` character that is not `IFS` white space, along with any adjacent `IFS` white space, shall delimit a field [. . .].
>
> c. Non-zero-length `IFS` white space shall delimit a field.

The 9-fields.2 revision of the elementary shell implements the necessary changes in the function `fieldsplit()`. With these rules, the attack you used in Exercise 3.1 will not work. However, you can still obtain a root shell from the `bad3suid` program (this is based on a real attack).

4 Countermeasures

The story above takes place in 1979. How effective are these types of attacks today?

Countermeasures have been introduced with more secure defaults and some tweaks to the behaviour of the most security-sensitive utilities. In most cases, however, users and programmers still need to be very careful.

4.1 Default value of **`PATH`**

Default initialization scripts and programs no longer put the current directory in `PATH`, nor do libraries provide an unsafe `PATH` if the variable is not explicitly set, as they used to do. If users really want to keep the current directory in their `PATH`, they should very carefully `ls` directories like `/tmp` *before* `cd`-ing into them. Putting the current directory last in `PATH` can also help, but it's not foolproof either: the attacker can put an `sl` in `/tmp` and wait for a user to mistype. Much better is not to put "`.`" or empty paths into `PATH` at all, and just use the "`./`" trick when we want to run a program that lives in the current directory.

Note, however, that the current directory may also enter `PATH` unintentionally. Empty paths can appear in `PATH` as a result of expanding undefined environment variables. Suppose you have installed a subsystem that puts its executables in a non-standard directory (a very common occurrence). You put the path to that directory in a variable, then expand that variable into your `PATH` in some of your shell initialization scripts:

```
    mybin=/opt/mysubsys/v0.1/bin
    # lots of other stuff
    PATH=$mybin:/usr/local/bin:/usr/bin:/bin
```

Some time later you uninstall the subsystem, delete the line that creates `mybin`, but forget to remove `$mybin` from the assignment to `PATH`. Now you have an empty path in your `PATH`.

4.2 **`IFS`** according to POSIX

Shells still implement `IFS`, but they use it in a much more restricted way. The 9-fields.3 revision of the elementary shell contains an implementation that adheres more strictly to what POSIX says. First of all, the shell must reset the value of `IFS` when it starts (section 2.5.3 of the standard). Second, field splitting must be applied only to the portions of the words that result from a previous expansion (point 2 of section 2.6 of the standard). For example, in a string like "`a,b,$X`", only the characters (if any) that result from the expansion of `$X` should be scanned for `IFS` separators. Assume that `X=c,d` and `IFS=,`: then field splitting of "`a,b,$X`" will produce two fields: "`a,b,c`" and "`d`". As a further example, the "`ls,-l`" string of Section 3.1 will not be split at all, since no expansion is called for.

4.3 Privilege drop

There is also a more general line of defense, that is implemented in all modern shells. If we try these attacks on a modern system, we will find that the resulting set-uid shell will not give us root access. For example, consider the `PATH` attack above, where we tried to create a set-uid shell disguised as a normal `hello.c` file. After successfully running the attack, we can confirm that the set-uid flag is set on `hello.c` , but when we run it we do not get the root prompt. If we run the `id` program we will see that our uid is still the unprivileged one that we already had. When they start, many shells (including `bash` and `dash`) call `getuid()` and `geteuid()` to get the real and the effective uids of the process that is running them. If the two uids are different, the shells will call `seteuid()` to reset the effective uid back to the real one. To implement this mitigation in our shell, we should add the following code to `main()`, before the program does anything else:

```
        seteuid(getuid());
```

(And similarly for the effective and real group).

We will see in a moment that this check is completely ineffective against the `PATH` attack of Section 2. This mitigation only addresses the case of set-uid programs calling `system()`. In fact, set-uid programs should *never* call `system()`, because shells, especially the modern ones, are large and complex programs with possibly many little-understood quirks and unexpected behavior. Here are just a few examples:

1. up to version 4.2-208, `bash` function names allowed the "/" character in them; an attacker could therefore easily redirect a `system("/bin/cmd")` by defining a `/bin/cmd` function;

2. up to version 4.4, `bash` "xtrace" feature would expand `PS4` before executing any command; xtrace can be set in `SHELLOPTS` and `PS4` can execute the attacker's payload using command substitution.

If an external program is really needed, it is better to use one of the `exec*()` functions without going through a shell. If a buggy set-uid program calls `system()` anyway[2], the shell will use the above mitigation to drop privileges and prevent harm.

In the case of the `PATH` attack, on the other hand, the above mitigation is just a minor inconvience for the attacker. Remember that the shell is not magic, and the attacker can create her own shell, one that does not check the uids. Better yet, we can note that `sudo` is able to give us a root prompt (`sudo -s` or `sudo -i`), of course if the system configuration allows it. Remember that even `sudo` is not magic: it is just set-uid root. So, when we run `sudo` our real uid is different from the effective uid. However, `sudo` is apparently able to hide this fact from any shell and avoid the privilege drop. How is this possible? The

[2]Or, as we will see later on, is forced to call `system()` by an attacker exploiting some bug.

solution is simple: if our effective uid is 0, we can call `setuid()` and set the *real* uid to 0 as well. Now the two ids are no longer different, and the shell's check becomes ineffective. This is (essentially) what `sudo` does before running the shell, and this is what we can do ourselves. We compile the following program:

```
#include <unistd.h>
int main()
{
    setuid(0);
    execl("/bin/sh", "/bin/sh", NULL);
}
```

and put it somewhere, say in `/home/attacker/mysudo`. Then we use the following payload for the `PATH` attack:

```
  chown root /home/attacker/mysudo
  chmod u+s  /home/attacker/mysudo
```

If root is caught in the `PATH` trap, she will turn our `mysudo` program into a set-uid program that will give us a root prompt.

Because it is so easy to defeat the (e)uid check, some shells don't even try to protect themselves in the general case. In `bash` and `dash`, for example, you can avoid going through the `mysudo` program above: create the set-uid root shell and pass it the `-p` option. The shell will skip the check and give you the root prompt. The `system()` use case should still be safe, however, since in this case the attacker cannot control the options that are passed to the shell at startup.