Entry

Jeremy Mates

June 27, 2021

```
$ FOO=a ruby -e 'puts ENV["FOO"], ARGV' b c
a
b
c
```

An important difference between the arguments list–b and c to the ruby script, or a longer list at the exec level–and the environment list–FOO=a and various unseen others–is that environment variables are inherited by child processes. This means placing a password or other secret data in an environment variable may be a poor practice. Environment variables can be difficult to whitelist, see sudo or doas for example code.

Use the `env(1)` command to inspect what environment variables are set, or language-specific features to view what is often the same list. Here are some different ways of doing the same thing.

```
$ export FOO=zot
$ expect -c 'puts $env(FOO); puts [exec env | grep FOO]'
zot
FOO=zot
$ perl -E 'say $ENV{FOO}; exec "env"' | grep zot
zot
FOO=zot
$ ruby -e 'puts ENV["FOO"]; exec "env"' | grep zot
FOO=zot
```

Note that the `expect` (TCL) `exec` is nothing like the OS `execv(3)` system call or the Perl or Ruby `exec` functions; the TCL `exec` is actually what is usually called `system(3)`. `system(3)` runs a shell command while the `execv(3)` replaces the current process with something else. Which to use depends on the language, and whether you want the original process to hang around afterwards. `system(3)` is often exploited if an attacker can pass in arbitrary shell code, somehow.

`ruby` needs a `STDOUT.flush` call to behave like the other processes. This is a buffering issue.

```
$ ruby -e 'puts ENV["FOO"]; STDOUT.flush; exec "env"' | grep zot
zot
FOO=zot
```

On linux, the `/proc/.../environ` file is another way to inspect the environment of a process.

```
# cd /proc/$$
# cat environ
USER=rootLOGNAME=rootHOME=/rootPATH=/usr/local/sbin:/us...
```

If the output is a mess, run it through a hex viewer–`hexdump(1)`, `od(1)` or with `vim` installed `xxd`–to see what is going on. This may reveal characters such as

```
# hexdump -C environ | head -1
00000000  55 53 45 52 3d 72 6f 6f  74 00 4c 4f ...
```

`nul` as shown by the hex code `00`. Now is a good time to mention `ascii(7)` which contains a handy chart of the ASCII characters; Unicode or other not-ASCII encodings can (sometimes) be identified by hex values north of `7F` and may need encoding-aware tools to better display and debug them. The `tr(1)` command should suffice to improve `environ` for human consumption; map the `NUL` values to newlines:

```
# tr '\0' '\n' < environ
...
```

Various invisible characters or Unicode characters that appear to be similar to ASCII but are not can also be problematic. A hex viewer should reveal their presence. The font will also influence how similar different characters appear to be. Even worse, terminals or operating systems may normalize characters run through copy and paste, or a document may contain something besides the indicated character. All this may complicate debugging. Take screenshots, use a hex dumper, and show the code used.

```
$ ls -al
ls: -al: No such file or directory
$ echo 'ls -al' | hexdump
00000000 6c 73 20 e2 80 93 61 6c 0a
```

Here the `e2 80 93` between the space `20` and the letter a `61` represent the Unicode character 'EN DASH' `U+2013` in UTF-8.

When something weird happens, run the data through a hex viewer, or use a process tracing tool such as `strace`. `strace` will reveal the exact argument list used and may help diagnose how an unknown ball of code is calling a program. This usually boils down to some `exec` call, though there may also be calls that involve the shell somehow.

```
$ strace -f -o eg1 ruby -e ''echo easygrep''
$ strace -f -o eg2 ruby -e ''echo easygrep&''
$ grep easygrep eg* | grep execve | grep -v ruby
eg1:4186 execve("/usr/bin/echo", ["echo", "easygrep"] ...
eg2:4206 execve("/bin/sh", ["sh", "-c", "echo easygrep&"] ...
```

With a single character change `ruby` switches from performing a direct exec of `echo` to wrapping the command in a shell call. This may be good to know if one had assumed the command was always run under the shell, or the other way around, and one or the other methods is broken for some reason. There can also be security implications.

On the Ordering of Arguments

Another complication with arguments is that programs may support subcommands where the arguments for one level of the command probably should not be mixed with those of the subcommand; other programs such as `find(1)` or `pw` on FreeBSD may require that various arguments appear in specific ways;

```
# wrong!
find -name "foo*" /etc

# correct
find /etc -name "foo*"

# same -C, different meanings
git -C <path> commit -C <commit> ...
```

this is in contrast to other commands where the options can appear after other arguments. Such usages may be frowned on, or unportable.

```
# atypical
$ grep root /etc -r
...

# more common
$ grep -r root /etc
...

# disable option processing, see getopt(3)
$ grep -r -- "$pattern" /etc
...
```

In general the manual page must be studied to see which inputs and in what order are supported; keep in mind that there are various historical warts, `find(1)` and especially `dd(1)` that use distinct grammars or otherwise do not fit the mold of more common programs.

Now let us take a closer look at environment variables, and how to

Duplicate Environment Variables

...but aren't environment variables stored in some sort of hash or associative array, which cannot by definition have duplicate keys? Yes. In some languages. Which are abstractions on top of what is actually a list, and lists may contain duplicate entries. Also there's literally nothing that distinguishes command line arguments from the environment variables. Some assembly and the judicious use of a debugger may help clear up this fog, or at least exchange the fog for the deep end of the pool.

```
bits 64
section .text
global  _start          ; linker entry point
_start: mov rbp,rsp     ; save pointer to stack pointer
        pop rcx         ; get argument count off stack
_nexta: pop rcx         ; loop popping off arg pointers
        cmp rcx,0       ; ... until 64-bit 0 value
        jne _nexta      ; GOTO!
_nexte: pop rcx         ; loop popping off env pointers
        cmp rcx,0       ; ... until 64-bit 0 value
        jne _nexte      ; GOTO!
        mov rax,60      ; sys_exit syscall
        mov rdi,0       ; exit code
        syscall         ; make it so
```

This is NASM format assembly for an AMD64 Linux system that does little more than pop 64-bit values off the stack into a register. This is only useful under a debugger. With this file saved as `envandargs.asm` the assembly, linking, and launching it under a debugger may run something like

```
$ nasm -f elf64 -g -F dwarf -o envandargs.o envandargs.asm
$ ld -o envandargs envandargs.o
$ gdb envandargs
...
```

Then, in `gdb`, we keep track of the `RCX` register:

```
(gdb) set args a bb ccc
(gdb) display /x $rcx
(gdb) break _start
Breakpoint 1 at 0x400080: file envandargs.asm, line 6.
(gdb) run
Starting program: /home/jmates/envandargs a bb ccc

Breakpoint 1, _start () at envandargs.asm:6
6       _start: mov rbp,rsp     ; save pointer to stack pointer
1: /x $rcx = 0x0
(gdb) next
7               pop rcx         ; get argument count off stack
1: /x $rcx = 0x0
(gdb) next
_nexta () at envandargs.asm:9
9       _nexta: pop rcx         ; loop popping off arg pointers
1: /x $rcx = 0x4
(gdb)
```

One confusion with `gdb` is that it shows the line of code it is about to run, not what it just executed.

The `pop rcx` has placed 4 into the `RCX` register; this is the argument count, or what would be available in `int main (int argc, ...` in a C program. The argument count includes the program name plus here the arguments `a`, `bb`, and `ccc`.

```
(gdb) next
_nexta () at envandargs.asm:10
10              cmp rcx,0       ; ... until 64-bit 0 value
1: /x $rcx = 0x7fffffffeef2
(gdb)
```

`RCX` should now contain the address of the first argument; let's inspect that, looking for a `s` string value–a list of not-`nul` characters followed by a `nul` character. Actually let's cheat and peek at several string values. . .

```
(gdb) x /s 0x7fffffffeef2
0x7fffffffeef2: "/home/jmates/envandargs"
(gdb) x /5s 0x7fffffffeef2
0x7fffffffeef2: "/home/jmates/envandargs"
0x7fffffffef0a: "a"
0x7fffffffef0c: "bb"
0x7fffffffef0f: "ccc"
0x7fffffffef13: "USER=jmates"
(gdb)
```

These are the arguments directly followed by the environment variables. Let's step back and take a look at the stack. The original stack pointer was saved into the `RBP` register as the very first instruction. Since this is a 64-bit system, we'll look at ten of what `gdb` calls `giant` or eight byte chunks:

```
(gdb) x /10xg $rbp
0x7fffffffed20: 0x0000000000000004 0x00007fffffffeef2
0x7fffffffed30: 0x00007fffffffef0a 0x00007fffffffef0c
0x7fffffffed40: 0x00007fffffffef0f 0x0000000000000000
0x7fffffffed50: 0x00007fffffffef13 0x00007fffffffef1f
0x7fffffffed60: 0x00007fffffffef2a 0x00007fffffffef92
(gdb)
```

There's the argument count 4 followed by the four addresses that point to the values of those four arguments, then the zero value that divides the arguments from the environment–the "nothing" I spoke of earlier–and then some of the addresses of the environment variables. `gdb` can also shell out to hex viewers; this is another way to show the contents of the arguments:

```
(gdb) dump binary memory ddd 0x00007fffffffeef2 0x00007fffffffef1f
(gdb) shell xxd ddd
...
(gdb) quit
```

All very fascinating. Though, how can we add a duplicate environment variable into the environment list?

High Level Language

C, a high level language,[a] uses the exact same stack of values as the assembly only with handy names attached to these addresses, plus a bunch of additional cruft.[b] Here is the file `envdup.c`.

```
/* envdup - wrapper script to create duplicate environment entries */

#include <err.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

extern char **environ;
```

[a]This term is relative.
[b]Ease and additional cruft are hallmarks of high level languages.

```
int main(int argc, char *argv[]) {
    char **newenv;
    int envcount = 0;

    if (argc < 2) errx(64, "Usage: envdup command [args ..]");

    newenv = environ;
    while (*newenv++ != NULL) envcount++;

    newenv = malloc(sizeof(char *) * (envcount + 3));
    if (newenv == NULL) err(1, "malloc failed");
    if (envcount > 0)
        memcpy(newenv, environ, sizeof(char *) * envcount);
    newenv[envcount]     = "FOO=one";
    newenv[envcount + 1] = "FOO=two";
    newenv[envcount + 2] = NULL;

    environ = newenv;
    argv++;
    execvp(*argv, argv);
    err(1, "exec failed '%s'", *argv);
}
```

This is similar to the shell script

```
#!/bin/sh
FOO=one FOO=two exec "$@"
```

only that the shell does not let you create duplicate environment variables.

```
$ FOO=one FOO=two env | grep FOO
FOO=two
$ CFLAGS=-g make envdup
cc -g envdup.c -o envdup
$ gdb ./envdup
...
```

Let us first poke around in the debugger and inspect what C calls argv.

```
(gdb) set args a bb ccc
(gdb) break main
Breakpoint 1 at 0x100000db6: file envdup.c, line 10.
(gdb) run
...
Breakpoint 1, main (argc=4, argv=0x7fff5fbff648) at envdup.c:10
10          int envcount = 0;
(gdb) x /10xg 0x7fff5fbff648
0x7fff5fbff648: 0x00007fff5fbff840 0x00007fff5fbff859
0x7fff5fbff658: 0x00007fff5fbff85b 0x00007fff5fbff85e
0x7fff5fbff668: 0x0000000000000000 0x00007fff5fbff862
0x7fff5fbff678: 0x00007fff5fbff87d 0x00007fff5fbff888
0x7fff5fbff688: 0x00007fff5fbff891 0x00007fff5fbff8cc
(gdb) x /5s 0x00007fff5fbff840
0x7fff5fbff840: "/Users/jmates/tmp/envdup"
0x7fff5fbff859: "a"
0x7fff5fbff85b: "bb"
0x7fff5fbff85e: "ccc"
0x7fff5fbff862: "_=/Users/jmates/tmp/envdup"
(gdb) quit
...
```

Exactly like the assembly code, and even the same format on both Linux assembly and here C code being run on Mac OS X. Now to run the program proper:

```
$ unset FOO
$ ./envdup env | grep FOO
FOO=one
FOO=two
```

Duplicate environment entries for FOO have been set, as reported by env(1). envdup can be used to test what other languages or programs do with duplicate environment entries. For instance, is the first or second duplicate selected?

```
$ unset FOO
$ ./envdup bash -c 'echo $FOO; exec env' | egrep 'one|two'
two
FOO=two
$ ./envdup zsh -c 'echo $FOO; exec env' | egrep 'one|two'
one
FOO=one
```

bash selects the last, and zsh the first, and the duplicate has been removed. What about other languages?

```
$ ./envdup expect -c 'puts $env(FOO);puts [exec env]' | egrep 'one|two'
one
FOO=one
FOO=two
$ ./envdup perl -E 'say $ENV{FOO};exec "env"' | egrep 'one|two'
one
FOO=one
$ ./envdup ruby -e 'puts ENV["FOO"];STDOUT.flush;exec "env"' | egrep 'one|two'
one
FOO=one
FOO=two
```

Consistent selection of the first duplicate, except that Ruby and TCL have not cleaned up the environment and pass the duplicates on to env. Perl instead matches zsh and calls env with the duplicate removed.

```
$ expect -v
expect version 5.45
$ perl -v | sed -n 2p
This is perl 5, version 32, subversion 1 (v5.32.1) built for amd64-openbsd
$ ruby -v
ruby 3.0.1p64 (2021-04-05 revision 0fb782ee38) [x86_64-openbsd]
```

Duplication of environment variables is not a trivial issue; see https://www.sudo.ws/repos/sudo/rev/d4dfb05db5d7 for the security patch to sudo(8). In particular, if only the first environment variable is sanitized bash as shown above will use the second variable. This could allow an attacker to specify a custom second PATH or various LD_* environment values that a subsequent bash process would use. "Practical UNIX & Internet Security"[1] mentions the duplicate environment variable attack;

> "examine the environment to be certain that there is only one instance of the variable: the one you set. An attacker can run your code from another program that creates multiple instances of an environment variable. Without an explicit check, you may find the first instance, but not the others; such a situation could result in problems later on." p.717

sudo was patched for this two decades later. Other software remains completely unpatched; one idea is that perhaps libc should maybe somehow handle the duplicates before the program enters main?

https://sourceware.org/bugzilla/show_bug.cgi?id=19749

This may be difficult given the differences between bash and other software with regard to which duplicate is passed through. . . what does your language do?

execv(3) versus system(3)

These calls are used to execute new programs. Understanding the difference between `exec(3)` and `system(3)` is important as `system(3)` uses the shell while passing a shell command to `exec(3)` may not fare so well. The difference is complicated by languages that call `system(3)` `exec`–TCL, as noted above. Also, PHP. Some software make both interfaces available. `git(1)` in particular offers both `GIT_SSH` and `GIT_SSH_COMMAND` environment variables:

```
$GIT_SSH_COMMAND takes precedence over $GIT_SSH, and is interpreted
by the shell, which allows additional arguments to be included.
$GIT_SSH on the other hand must be just the path to a program (which
can be a wrapper shell script, if additional arguments are needed).
```

Less well documented programs may need a source code dive or `strace` to determine which is used. A shell command "not working" is a good indication that `exec(3)` is being used instead of `system(3)`. A minimal `exec(3)` example would be the following C code,

```
#include <err.h>
#include <unistd.h>
int main(int argc, char *argv[]) {
    argv++;
    execvp(*argv, argv);
    err(1, "exec failed");
}
```

which can illustrate the "`system` was assumed but `exec` used" problem:

```
$ make minexec
gcc     minexec.c   -o minexec
$ ./minexec echo 'hi there'
hi there
$ ./minexec 'echo hi there'
minexec: exec failed: No such file or directory
```

`strace` is most useful when a program omits details necessary to debug an issue; what is actually being run for the second command is the string `echo hi there` which typically does not exist in `PATH`. No shell word split is done by `execv(3)`; the string given as the first argument is searched for in `PATH`. `system(3)` by contrast will pass a string to `sh`. For a minimal C example the list of arguments must be converted into a single string, ideally with more error checking and fewer assumptions than are made here–this is something like `tr '\0' ' '` for the arguments list.

```
#include <stdlib.h>
#include <string.h>
int main(int argc, char *argv[]) {
    char *ap;
    argv++;
    argc -= 2;
    ap = *argv;
    while (argc--) {
        ap  = strchr(ap, '\0');
        *ap = ' ';
    }
    system(*argv);
}
```

This code by contrast does run `'echo hi there'` without error

```
$ make minsys
cc -O2 -pipe    -o minsys minsys.c
$ ./minsys 'echo hi here'
hi here
```

as what is being run is `/bin/sh -c 'echo hi there'`–compare the programs with `strace`. Redirections and other shell constructs are likewise acceptable to `system`, but not `exec`.

Maximum Arguments

There exist various limits on what a unix program will accept. These limits have been raised over time. The usual encounter will run something like

```
$ ls *
```

in a directory with a very large number of files, or files with very long names, or where many environment variables have been set; the shell first expands the glob, and then calls some `exec` function with the complete list of filenames. A typical limit is `ARG_MAX` that includes both the arguments and the environment list.

```
$ getconf ARG_MAX
2097152
$ mkdir big && cd big
$ perl -e '$n = "a"x128; $c = 16384;' \
  -e 'while ($c--) { `touch $n`; $n++ }'
$ ls *
-bash: /bin/ls: Argument list too long
```

Hence `xargs(1)`; this converts arbitrary amounts of standard input into as many as necessary executions of a command with the standard input converted to not more than the maximum allowed arguments.

```
$ ls | xargs echo | wc -l
17
```

Seventeen `echo` executions were necessary to get through the list. Note that parsing `ls` is typically a bad idea; filenames really should be passed around `nul`-delimited, or handled internally by a process. Modern versions of `find` support either the `-exec ... {} +` form to act like `xargs` without the problems of `xargs`, or to simply remove all these files there is the more efficient `-delete` flag.

```
find . -mindepth 1 -exec rm {} +
find . -mindepth 1 -delete
```

`ARG_MAX` is not be the only limit; another one involves interpreted scripts, those with a "hash bang" or "shebang" line. `zsh` 5.3.1 defaulted to a static `POUNDBANGLIMIT` of 64 characters in `Src/exec.c`. For a typical shebang line of

```
#!/usr/bin/env expect
puts "hello world"
```

this is not a problem, though packages in NixOS are particularly adept at running past this limit.

```
# find /nix/store -type f -exec perl -nE \
  'say length if /^#!/ and length > 64; close ARGV' {} +
68
66
66
...
# find /nix/store -type f -exec perl -nE \
  'say length if /^#!/ and length > 64; close ARGV' {} + \
  | sort -nr | head -1
1466
#
```

A solution? Increase the size of that buffer in `zsh`. I probably should talk more about buffers, but that's another article. . .

References

1. Simson Garfinkel and Gene Spafford. *Practical UNIX & Internet Security, Second Edition*. O'Reilly & Associates, Inc., 1996.