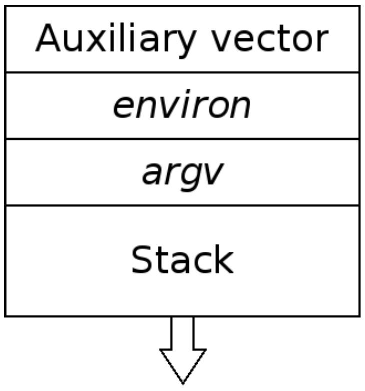
getauxval() and the auxiliary vector

By **Michael Kerrisk**
October 10, 2012

There are many mechanisms for communicating information between user-space applications and the kernel. System calls and pseudo-filesystems such as `/proc` and `/sys` are of course the most well known. Signals are similarly well known; the kernel employs signals to inform a process of various synchronous or asynchronous events—for example, when the process tries to write to a broken pipe or a child of the process terminates.

There are also a number of more obscure mechanisms for communication between the kernel and user space. These include the Linux-specific netlink sockets and user-mode helper features. Netlink sockets provide a socket-style API for exchanging information with the kernel. The user-mode helper feature allows the kernel to automatically invoke user-space executables; this mechanism is used in a number of places, including the implementation of control groups and piping core dumps to a user-space application.

The auxiliary vector, a mechanism for communicating information from the kernel to user space, has remained largely invisible until now. However, with the addition of a new library function, `getauxval()`, in the GNU C library (glibc) 2.16 release that appeared at the end of June, it has now become more visible.



Historically, many UNIX systems have implemented the auxiliary vector feature. In essence, it is a list of key-value pairs that the kernel's ELF binary loader (`fs/binfmt_elf.c` in the kernel source) constructs when a new executable image is loaded into a process. This list is placed at a specific location in the process's address space; on Linux systems it sits at the high end of the user address space, just above the (downwardly growing) stack, the command-line arguments (*argv*), and environment variables (*environ*).

From the description and diagram, we can see that although the auxiliary vector is somewhat hidden, it is accessible with a little effort. Even without using the new library function, an application that wants to access the auxiliary vector merely needs to obtain the address of the location that follows the `NULL` pointer at the end of the environment list. Furthermore, at the shell level, we can discover the auxiliary vector that was supplied to an executable by setting the `LD_SHOW_AUXV` environment variable when launching an application:

```
$ LD_SHOW_AUXV=1 sleep 1000
AT_SYSINFO_EHDR: 0x7fff35d0d000
AT_HWCAP:        bfebfbff
AT_PAGESZ:       4096
AT_CLKTCK:       100
AT_PHDR:         0x400040
```

```
    AT_PHENT:         56
    AT_PHNUM:         9
    AT_BASE:          0x0
    AT_FLAGS:         0x0
    AT_ENTRY:         0x40164c
    AT_UID:           1000
    AT_EUID:          1000
    AT_GID:           1000
    AT_EGID:          1000
    AT_SECURE:        0
    AT_RANDOM:        0x7fff35c2a209
    AT_EXECFN:        /usr/bin/sleep
    AT_PLATFORM:      x86_64
```

The auxiliary vector of each process on the system is also visible via a corresponding `/proc/PID/auxv` file. Dumping the contents of the file that corresponds to the above command (as eight-byte decimal numbers, because the keys and values are of that size on the 64-bit system used for this example), we can see the key-value pairs in the vector, followed by a pair of zero values that indicate the end of the vector:

```
    $ od -t d8 /proc/15558/auxv
    0000000                    33      140734096265216
    0000020                    16           3219913727
    0000040                     6                 4096
    0000060                    17                  100
    0000100                     3              4194368
    0000120                     4                   56
    0000140                     5                    9
    0000160                     7                    0
    0000200                     8                    0
    0000220                     9              4200012
    0000240                    11                 1000
    0000260                    12                 1000
    0000300                    13                 1000
    0000320                    14                 1000
    0000340                    23                    0
    0000360                    25      140734095335945
    0000400                    31      140734095347689
    0000420                    15      140734095335961
    0000440                     0                    0
    0000460
```

Scanning the high end of user-space memory or `/proc/PID/auxv` is a clumsy way of retrieving values from the auxiliary vector. The new library function provides a simpler mechanism for retrieving individual values from the list:

```
    #include <sys/auxv.h>

    unsigned long int getauxval(unsigned long int type);
```

The function takes a key as its single argument, and returns the corresponding value. The glibc header files define a set of symbolic constants with names of the form `AT_*` for the key value passed to `getauxval()`; these names are exactly the same as the strings displayed when executing a command with `LD_SHOW_AUXV=1`.

Of course, the obvious question by now is: what sort of information is placed in the auxiliary vector, and who needs that information? The primary customer of the auxiliary vector is the dynamic linker (`ld-linux.so`). In the usual scheme of things, the kernel's ELF binary loader constructs a process image by loading an executable into the process's memory, and likewise loading the dynamic linker into memory. At this point, the dynamic linker is ready to take over the task of loading any shared libraries that the program may need in preparation for handing control to the program itself. However, it lacks some pieces of information that are essential for these tasks: the location of the program inside the virtual address space, and the starting address at which execution of the program should commence.

In theory, the kernel could provide a system call that the dynamic linker could use in order to obtain the required information. However, this would be an inefficient way of doing things: the kernel's program loader already has the information (because it has scanned the ELF binary and built the process image) and knows that the dynamic linker will need it. Rather than maintaining a record of this information until the dynamic linker requests it, the kernel can simply make it available in the process image at some location known to the dynamic linker. That location is, of course, the auxiliary vector.

It turns out that there's a range of other information that the kernel's program loader already has and which it knows the dynamic linker will need. By placing all of this information in the auxiliary vector, the kernel either saves the programming overhead of making this information available in some other way (e.g., by implementing a dedicated system call), or saves the dynamic linker the cost of making a system call, or both. Among the values placed in the auxiliary vector and available via `getauxval()` are the following:

- `AT_PHDR` and `AT_ENTRY`: The values for these keys are the address of the ELF program headers of the executable and the entry address of the executable. The dynamic linker uses this information to perform linking and pass control to the executable.

- `AT_SECURE`: The kernel assigns a nonzero value to this key if this executable should be treated *securely*. This setting may be triggered by a Linux Security Module, but the common reason is that the kernel recognizes that the process is executing a set-user-ID or set-group-ID program. In this case, the dynamic linker disables the use of certain environment variables (as described in the `ld-linux.so(8)` manual page) and the C library changes other aspects of its behavior.

- `AT_UID`, `AT_EUID`, `AT_GID`, and `AT_EGID`: These are the real and effective user and group IDs of the process. Making these values available in the vector saves the dynamic linker the cost of making system calls to determine the values. If the `AT_SECURE` value is not available, the dynamic linker uses these values to make a decision about whether to handle the executable securely.

- `AT_PAGESZ`: The value is the system page size. The dynamic linker needs this information during the linking phase, and the C library uses it in the implementation of the `malloc` family of functions.

- `AT_PLATFORM`: The value is a pointer to a string identifying the hardware platform on which the program is running. In some circumstances, the dynamic linker uses this value in the interpretation of `rpath` values. (The `ld-linux.so(8)` man page describes `rpath` values.)

- `AT_SYSINFO_EHDR`: The value is a pointer to the page containing the Virtual Dynamic Shared Object (VDSO) that the kernel creates in order to provide fast implementations of certain system calls. (Some documentation on the VDSO can be found in the kernel source file `Documentation/ABI/stable/vdso`.)

- `AT_HWCAP`: The value is a pointer to a multibyte mask of bits whose settings indicate detailed processor capabilities. This information can be used to provide optimized behavior for certain library functions. The contents of the bit mask are hardware dependent (for example, see the kernel source file `arch/x86/include/asm/cpufeature.h` for details relating to the Intel x86 architecture).

- `AT_RANDOM`: The value is a pointer to sixteen random bytes provided by the kernel. The dynamic linker uses this to implement a stack canary.

The precise reasons why the GNU C library developers have chosen to add the `getauxval()` function now are a little unclear. The commit message and NEWS file entry for the change were merely brief explanations of what the change was, rather than why it was made. The only clue provided by the implementer on the libc-alpha mailing list suggested that doing so was useful to allow for "future enhancements to the AT_ values, especially target-specific ones." That comment, plus the observation that the glibc developers tend to be rather conservative about adding new interfaces to the ABI, suggest that that they have some interesting new user-space uses of the auxiliary vector in mind.

(Log in to post comments)

getauxval() and the auxiliary vector
Posted Oct 11, 2012 21:20 UTC (Thu) by **luto** (guest, #39314) [Link]

Can glibc also give me a good random seed? Take AT_RANDOM, use it to seed some simple PRNG, keep the first chunk of output for a stack canary, and give me the rest. (The PRNG could be as simple as using an MGF or some large-output hash. Something like SHA-3 could be used for this purpose, since Keccak can produce any amount of output, sequentially.)

Currently AFAIR glibc zeros AT_RANDOM after using it for its own nefarious purposes.

Reply to this comment

getauxval() and the auxiliary vector
Posted Oct 12, 2012 12:03 UTC (Fri) by **njwhite** (guest, #51848) [Link]

(with apologies for the embarassing levels of ignorance this likely displays)

Could other applications read your auxiliary vector, and hence the seed you're using? Would this then have much effect on the guessability of subsequent random numbers?

Reply to this comment

getauxval() and the auxiliary vector
Posted Oct 12, 2012 16:44 UTC (Fri) by **mina86** (subscriber, #68442) [Link]

I don't think that's a real problem. The random bytes are not available to outside applications (unless they trace it like debuggers or strace do, but in this case, it would be easier to just read the random seed or just overwrite the seed).

Reply to this comment

getauxval() and the auxiliary vector
Posted Oct 12, 2012 16:42 UTC (Fri) by **mina86** (subscriber, #68442) [Link]

With this piece of code: https://gist.github.com/3880154 and glibc 2.11 I'm able to get some value of AT_RANDOM. Checking whether glibc simply leaves it be or it fills it with some other random data is left as an exercise for the reader. ;)

Reply to this comment

HWCAP human readable on my 32-bit Centrino
Posted Oct 11, 2012 21:46 UTC (Thu) by **pr1268** (subscriber, #24648) [Link]

I tried `sleep` with `LD_SHOW_AUX=1` set, and my `AT_HWCAP` values were the human-readable processor capabilities I get when I `cat /proc/cpuinfo`. Plus, I had an extra entry at the beginning, `AT_SYSINFO`.

FWIW I have GLIBC version 2.13 (which implies the `getauxval()` call isn't available to me). Just sharing my experiences here; thanks for the article!

Reply to this comment

getauxval() and the auxiliary vector
Posted Oct 15, 2012 17:48 UTC (Mon) by **dashesy** (guest, #74652) [Link]

Sorry if I am outright clueless, but it would be very interesting if it can be used for multiarch scenarios. Maybe to simplify mixing x64 and x86 in a distribution-agnostic fashion, or even more interesting; Not only to run the foreign code but also link to a library of foreign architecture.

Reply to this comment

getauxval() and the auxiliary vector
Posted Oct 15, 2012 19:21 UTC (Mon) by **BenHutchings** (subscriber, #37955) [Link]

> Sorry if I am outright clueless, but it would be very interesting if it can be used for multiarch scenarios.

Multiarch and biarch requite that each coexisting architecture has a distinct dynamic linker.

> Maybe to simplify mixing x64 and x86 in a distribution-agnostic fashion, or even more interesting; Not only to run the foreign code but also link to a library of foreign architecture.

This is not possible even with thunking of function calls (very expensive if you have to make 32/64-bit mode switches) as the different architectures have different memory layouts.

Reply to this comment

getauxval() and the auxiliary vector
Posted Nov 13, 2012 11:09 UTC (Tue) by **oak** (guest, #2786) [Link]

Nice. Yet another way for programs to find out on what kind of a platform they're running at runtime.

(If everything would use it instead of reading the /proc/self/auxv file directly, it would be nice, now it just means that tools like SB2 and user-space Qemu have yet another thing they need to catch & override, but at least this new method is easier to override.)

Reply to this comment

erratum: x86 cpufeatures location
Posted Dec 29, 2017 19:43 UTC (Fri) by **georg.s** (guest, #110733) [Link]

Actually, the x86 details are defined in

arch/x86/include/asm/cpufeatures.h

and not in

arch/x86/include/asm/cpufeature.h (sic!)

See also e.g.:

https://git.kernel.org/pub/scm/linux/kernel/git/stable/li...

Reply to this comment