Linux MIPS - A soft target: past, present, and future

Cyber-ITL

Researched and written by: Parker Thompson, Mudge Zatko

{parker,mudge}@cyber-itl.org

Introduction

The mission of the CITL (Cyber Independent Testing Laboratories), a 501(c)3 non-profit organization, is to quantify and measure security hygiene in software. More specifically we have a focus on measuring the existence and coverage of security artifacts in the build process and the resulting strengths/weaknesses during execution. Through this we hope to bring transparency to security practices in software we all use and consume so that everyone can make better informed decisions as software consumers.

To date CITL has spent significant time working on measurements of desktop software, primarily x86/x64 and ARM, across Windows, OS X, and Linux. Recently CITL has been expanding scope to include other targets, such as firmware images from embedded systems.

During this work we discovered an issue with MIPS systems running Linux. This could have major implications for a variety of devices such as home routers, large scale infrastructure switches/routers, government communications, and security gear.

For the period from 2001 to the present most of these systems lack basic stack based Data Execution Prevention (DEP). This appears to be the case even after two patches were introduced in 2016, one of which unintentionally introduced a way to bypass Address Space Layout Resolution (ASLR). We also observe a significant lag in adoption of the latest linux kernels, and related compiler toolchains, in many MIPS devices including end user devices.

The long lived lack of basic stack DEP combined with the new security exposure (a mitigation bypass) introduced two years ago, the identification of the continuing related vulnerability introduced by the 2016 patches, and the slow adoption of kernels toolchains, leads us to believe that Linux MIPS systems will remain a soft target to exploit for years to come.

This paper discusses the general problem we noticed with LINUX MIPS and the problem's origins, provides a brief background on the technical terms involved, and then provides a deep dive on the problem's discovery and implications.

A companion paper takes a specific look at how this problem impacts home routers, a product category where MIPS is particularly common. The results show broad lack of security fundamentals across all of the Linux MIPS routers.

Problems and Origins

We trace the origin of missing stack based DEP, and the new ASLR mitigation bypass, ultimately back to ambiguity in the MIPS specification and how Linux had to handle that ambiguity. The specification does not mandate all behaviors for the Floating Point Unit (FPU) instructions/operations, so hardware implementations from different vendors were not consistent in their behavior. To address this, the Linux Kernel emulates some instructions in an attempt to normalize FPU behavior across different hardware implementations.[1]

Due to differences in processor implementation, and for technical reasons which we explore below, the emulated floating point code is placed on the local thread stack and executed. An executable stack is a prerequisite for basic stack based buffer overflows and has been a well known security liability since roughly 1995.

In 2003, Microsoft introduced DEP for program stacks. Similar features have been a part of the Linux Kernel Mainline since 2.6.8, which was released in 2004.

However, Linux MIPS did not receive this ability until 2016. This means that from 2001 to 2016, the norm for Linux MIPS stacks was to be marked as executable.

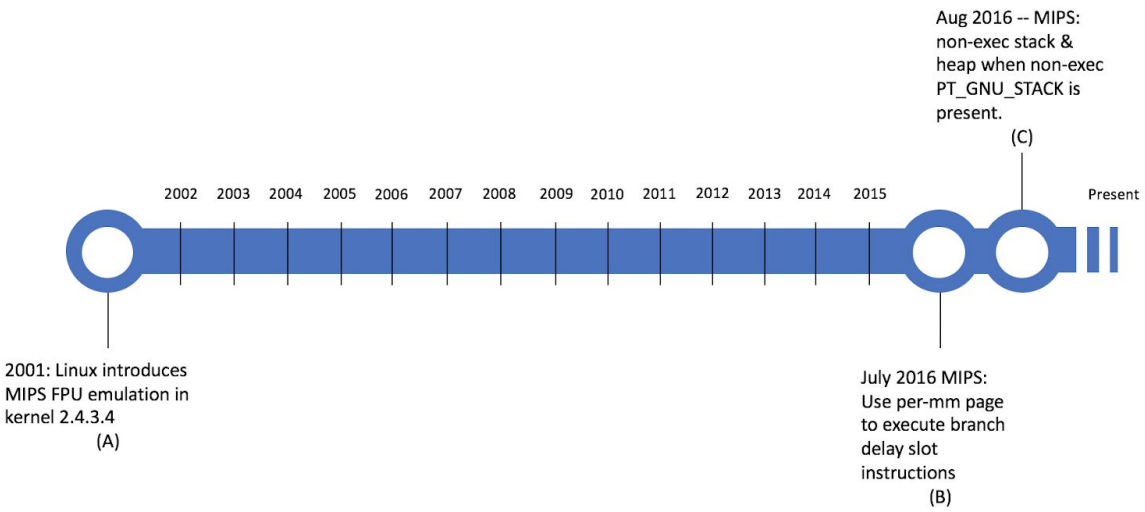
In July 2016, a patch was introduced to move code off of the stack for Linux MIPS. Later in August 2016, a patch was introduced to allow the kernel to mark the stack and heap as non-execute if requested by an application.[2]

Unfortunately, the July 2016 patch moving code execution away from the stack created a new segment at a predictable location that was marked as read, write, and execute. This provides a core primitive with the ability to bypass DEP *and* ASLR.

---

[1] An overview of what the kernel is attempting to achieve by emulating FPU instructions may be found at https://www.linux-mips.org/wiki/Floating_point%23The_Linux_kernel_and_floating_point

[2] Of course, this "opt-in" approach meant that compilers and toolchains would need to be modified to create applications that requested a non-executable stack, as the default behavior continued to provide an executable stack.



Timeline Key:

(A) 2001: Linux introduces FPU emulation in kernel 2.4.3.4. This puts code on the stack and executes it there requiring the stack be marked as readable, writable, and executable.
(B) 2016 July: a new page was introduced to execute branch delay slot instructions. This was introduced to remove the code being inserted and executed on the program stack. However, this fix introduced a new fixed location segment that can be used to bypass ASLR defenses.[3]
(C) 2016 August: a patch to make the stack and heap non-execute was introduced, if a PT_GNU_STACK was present. However, as noted in the patch none of the toolchains used to build executables created a PT_GNU_STACK and the stack would remain executable until this was addressed in compiler toolchains.[4]

In summary, Linux MIPS binaries have been easier to exploit by way of classic stack overflow attacks for over a decade, and continue to be so according to our examination of toolchain patches. Additionally, the fix that moved FPU emulation off the stack created a separate DEP *and* ASLR exposure. Even if patches were introduced today for both the kernel and the toolchains, the lag in adoption implies it could be years before Linux MIPS devices can be expected to have basic DEP and ASLR.

What is MIPS, what are ASLR and DEP, and why do they matter?

MIPS is a RISC CPU architecture that was introduced over 30 years ago and is still in use today across communications and security systems, routers, and a range of other devices. MIPS chips

[3] https://github.com/torvalds/linux/commit/432c6bacbd0c16ec210c43da411ccc3855c4c010
[4] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=1a770b85c1f1c1ee37afd7cef5237ffc4c970f04

were used in high-end desktops (SGI) and MIPS was among the first architectures targeted by windows NT (before Intel i386). Presently MIPS chips are found in a large number of home routers and even high end Next Generation Firewalls (e.g. those made by Palo Alto Networks). While there are a few specific markets in which MIPS chips are still common, the most common place a consumer will interact with a likely be a device running on a MIPS chip will be in a embedded or IoT device. Even with the recent growth of ARM chips in the embedded device market, MIPS chips are still quite common. One of the most common classes of device to still use MIPS chips are home routers. In a survey of popular home routers , 8 of 14 devices use MIPS CPUs. More on the impacts of this particular security problem on those devices can be found in our companion paper[5].

ASLR, which stands for Address Space Layout Randomization, randomizes memory locations within a process. Some classes of attack rely upon certain code or data being at predictable locations. ASLR is a well-established defensive technique used to prevent memory corruption exploits from functioning correctly.

DEP, which stands for Data Execution Prevention, aims to mark certain data and data locations as non-executable. This is done in an attempt to prevent attackers from providing their own code to execute when hijacking control of a target application. The most rudimentary part of DEP is marking a program's stack segment as non-executable to prevent basic stack-based buffer overflow payloads. Like ASLR, DEP is an industry-standard security practice which has been used to prevent memory corruption exploits for years.

One element of DEP is ensuring that the stack is not executable. In Linux, this means having the PT_GNU_STACK permissions specified as RW (read write). If the permissions aren't specified, or the permissions include an X, then the stack is executable.

ASLR and DEP work in tandem to protect against memory corruption attacks. The loss of either mitigation significantly reduces the overall efficacy of memory corruption defenses; absence of both is an even more serious exposure. Both of these mitigations have been available in modern operating systems (e.g. Linux, Windows, macOS), and architectures other than MIPS, for well over a decade as mentioned above.

The next section explains how this unexpected absence in MIPS was caused by code in the Linux kernel.

The security problem in a nutshell

The issue involves how the Linux kernel loads and executes ELF files. The interactions between the kernel and compiler toolchains can be complex. The kernel needs to be able to handle binaries with and without aPT_GNU_STACK. The compilers need to know how best to generate

[5] https://cyber-itl.org/assets/papers/2018/build_safety_of_software_in_28_popular_home_routers.pdf

default ELF files so they will run successfully on the majority of kernel versions. This interaction causes a complex set of scenarios where security bypasses can be present, and where security is sometimes sacrificed in the name of broader compatibility. Below are two charts. The first chart breaks down different kernel versions and how the existence, or lack of, a stack segment can impact the runtime. The second chart shows the default output and permissions of the PT_GNU_STACK segment of GCC and clang.

Kernel Version	Stack Default without PT_GNU_STACK (RW)	Stack Default with PT_GNU_STACK (RW)	ASLR bypass introduced?	Stack DEP	Note
2.4.3.4 thru 4.7	Executable	N/A* - Not compatible with these versions	No	No	*Specific code to handle this case was not introduced until 4.8. A binary with a no-exec PT_GNU_STACK would likely crash with FPU emulation code being on the stack.
4.8 and above	RWX	RW	Bypass	Bypass*	*The stack will be non-executable, but a new mapping will be at a static address and read-write-execute

Because the presence of the PT_GNU_STACK segment affects how the kernel loads the process, below is a table of two different compilers and how they behave with Linux MIPS binaries.

Compiler	Outputs PT_GNU_STACK by default	Permissions
GCC (6.3.0)	False	N/A
GCC (8.3.0 - from unstable branch)	False (* see paragraph below)	N/A
Clang (3.8.1-24)	True	RWX

It should be noted that as of writing this, 6.3.x is the version of GCC MIPS that is available through the main distro package managers Ubuntu, Debian, etc., for Intel/AMD and MIPS. This means that the vast majority of product manufacturers and developers will be using this toolchain. The GCC 8.2.0 MIPS compiler toolchain packages were only available in unstable and experimental branches for our Linux systems.

Assuming a vendor builds their commercial products using unstable/experimental branches for their development environments, GCC 8.2 MIPS defaulted to an executable stack. We found it is possible to specify a non-executable stack explicitly to the linker, which then correctly outputs a GNU_STACK RW segment.

The stable package of GCC MIPS cross-compilers for Linux distributions is GCC 6.3.0. For GCC compilers we note two challenges: 1) the stable package version being distributed and used is 6.3, and 2) the unstable/experimental 8.2 MIPS packages default to an executable stack on MIPS binaries. These issues make the fact that GCC 8.2 *can* output the correct stack permissions relatively moot in practice[6].

What products do I use that might be impacted?

For home routers running Linux MIPS an executable stack and/or the fix, which introduces a new DEP and ASLR bypass, lowers the cost to attackers to leverage bugs that may have otherwise been non-exploitable or costly to exploit reliably. Lower cost targets are more attractive to a range of adversaries. From the consumer reports article[7], CITL compiled a list of 14 popular home routers, 10 of which had firmware available to the public online. Of the 10 firmware images we retrieved, 70% of the devices were MIPS based. Analysis of these MIPS based home routers is in the companion paper.

High end systems in corporate or communications roles running Linux on MIPS may also exhibit this security deficiency. Some examples of systems advertising that they are built on MIPS processors, for both their data and management backplanes, include Next Generation Firewalls such as Palo Alto Networks {PA-200, PA-500, PA-2000}[8].

---

[6] A GCC 8 cross compiler package is listed in the unstable branch for Debian with dependencies that did not allow even our unstable and experimental branch systems to correctly retrieve and install the package correctly. Across hundreds of current (as of September 2018) firmware images we note that binaries for home routers, unsurprisingly, have been built with toolchains comparable or earlier than 6.3.0. Further we noticed, as we mention below, that the same compiler toolchains we testing natively within a MIPS environment and being used to cross-compile to MIPS targets produced different PT_GNU_STACK outputs in the case of Clang.

[7] https://www.consumerreports.org/products/wireless-routers/ratings-overview/

[8] See, for example, this Common Criteria Evaluation (CCE) for Palo Alto products: https://www.commoncriteriaportal.org/files/epfiles/st_vid10392-add1.pdf. It is telling that the aforementioned CCE does not consider differences in processors as a security-relevant issue.

Both the issue presented in this paper for Linux on MIPS and recent processor side channel issues such as Meltdown and Spectre help highlight that CCE focuses more on design processes rather than the security of the end product. We believe this is a common problem across many evaluations that imply “security”, but are focused much more on backend processes and uniformity (see section “Changes to TOE:”, item 4, in the above link).

Regarding Palo Alto Network products, we have not evaluated them. We are using them as an example of types of products that are Linux based and have MIPS architectures that could suffer from these issues. We hope vendors explain how they have addressed or mitigated these issues or otherwise provide transparency so that customers may make informed choices or otherwise check for themselves.

How to check your system

Examining the ELF program headers in binaries on the system will reveal the presence or absence of a PT_GNU_STACK segment. If this segment is *not* present the kernel will assume READ_IMPLIES_EXEC and mark the stack, and other mappings, as read, write, and execute. The program `readelf` can be used to show the program headers of binaries.

```
user@debian:~$ readelf -l /bin/cat

Elf file type is DYN (Shared object file)
Entry point 0x2450
There are 11 program headers, starting at offset 52

Program Headers:
 Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
 PHDR           0x000034 0x00000034 0x00000034 0x00160 0x00160 R E 0x4
 INTERP         0x000194 0x00000194 0x00000194 0x0000d 0x0000d R   0x1
 ABIFLAGS       0x0001c8 0x000001c8 0x000001c8 0x00018 0x00018 R   0x8
 REGINFO        0x0001e0 0x000001e0 0x000001e0 0x00018 0x00018 R   0x4
 LOAD           0x000000 0x00000000 0x00000000 0x081b4 0x081b4 R E 0x10000
 LOAD           0x008f28 0x00018f28 0x00018f28 0x0039c 0x00528 RW  0x10000
 DYNAMIC        0x00021c 0x0000021c 0x0000021c 0x00120 0x00120 R   0x4
 NOTE           0x0001a4 0x000001a4 0x000001a4 0x00020 0x00020 R   0x4
 NOTE           0x0001f8 0x000001f8 0x000001f8 0x00024 0x00024 R   0x4
 GNU_RELRO      0x008f28 0x00018f28 0x00018f28 0x000d8 0x000d8 R   0x1
 NULL           0x000000 0x00000000 0x00000000 0x00000 0x00000     0x4
```

In the above we notice an *absence* of PT_GNU_STACK in the list of Program Headers column. If you have shell access you can perform this check on the system directly. Otherwise, you may need to take other steps to extract binaries from the system and examine them manually or with an application like `readelf` elsewhere.

If you have do have shell access on the target system you can see the effect at runtime looking at the /proc entries. Here we view the /proc entries for the same binary as before.

```
user@debian:~$ cat /proc/self/maps
559be000-559c7000 r-xp 00000000 08:01 20         /bin/cat
559d6000-559d7000 r-xp 00008000 08:01 20         /bin/cat
559d7000-559d8000 rwxp 00009000 08:01 20         /bin/cat
55a04000-55a25000 rwxp 00000000 00:00 0          [heap]
77985000-779a7000 rwxp 00000000 00:00 0
779a7000-77b42000 r-xp 00000000 08:01 261167     /usr/lib/locale/locale-archive
77b42000-77cae000 r-xp 00000000 08:01 130656     /lib/mipsel-linux-gnu/libc-2.24.so
77cae000-77cbe000 ---p 0016c000 08:01 130656     /lib/mipsel-linux-gnu/libc-2.24.so
77cbe000-77cc1000 r-xp 0016c000 08:01 130656     /lib/mipsel-linux-gnu/libc-2.24.so
77cc1000-77cc4000 rwxp 0016f000 08:01 130656     /lib/mipsel-linux-gnu/libc-2.24.so
77cc4000-77cc6000 rwxp 00000000 00:00 0
77cc6000-77ce9000 r-xp 00000000 08:01 130652     /lib/mipsel-linux-gnu/ld-2.24.so
77ced000-77cef000 rwxp 00000000 00:00 0
77cf3000-77cf4000 r-xp 0019a000 08:01 261167     /usr/lib/locale/locale-archive
77cf4000-77cf6000 rwxp 00000000 00:00 0
77cf6000-77cf7000 r--p 00000000 00:00 0          [vvar]
77cf7000-77cf8000 r-xp 00000000 00:00 0          [vdso]
77cf8000-77cf9000 r-xp 00022000 08:01 130652     /lib/mipsel-linux-gnu/ld-2.24.so
77cf9000-77cfa000 rwxp 00023000 08:01 130652     /lib/mipsel-linux-gnu/ld-2.24.so
7fae9000-7fb0a000 rwxp 00000000 00:00 0          [stack]
7ffff000-80000000 rwxp 00000000 00:00 0
```

In the above output we see that not only is the stack marked as executable, but so is the heap and other mappings, shown in red.

If your system is running a Linux kernel version 4.8 or higher we recommend checking for the emulation page in the maps. Again with shell access on the system, `cat /proc/self/maps` and look for a single executable anonymous mapping. On 32 bit systems it should be located at 0x7ffff000 and on 64 bit systems it should be 0xfffffff000. An example of the proc entries is shown below,marked in red. If your system has the mapping but it is at a different address, it is possible your system is using a non-standard page size. In the following output we note the fix to the executable stack, but also the creation of the new emulation segment as readable, writable, and executable at the expected fixed address range.

```
00400000-00407000 r-xp 00000000 08:01 209        /home/user/test_stack
00416000-00417000 rwxp 00006000 08:01 209        /home/user/test_stack
77b0d000-77b0e000 r--p 00000000 00:00 0          [vvar]
77b0e000-77b0f000 r-xp 00000000 00:00 0          [vdso]
7fae4000-7fb05000 rw-p 00000000 00:00 0          [stack]
7ffff000-80000000 rwxp 00000000 00:00 0
```

How did CITL find this? Why didn't people know about this sooner?

The absence of these basic security defenses is invisible to the developer unless measurements of the resulting binaries are performed. Such measurements are not standardly done in the industry. CITL is the first organization to measure the usage of security measures in consumer software products at scale and we came across this surprising omission in the course of our work. While it is well known exploits of the past took significant advantage of an executable stack, and modern exploits often need to bypass ASLR, we believe that this significant degradation in basic security (and the introduction of a DEP and ASLR bypass in the patched versions) in many operational environments is largely unknown to consumers, developers, and security researchers. It is apparent that this issue is at least partially understood by some of the kernel developers and compiler writers based upon comments in their code, as we will show below, however it is unknown to us if the full security ramifications were considered.

In the process of expanding the CITL binary analysis tools that measure security hygiene in software to include MIPS targets, the MIPS binaries we collected for Debian (9.4, kernel: 4.9.0-6-4kc-malta) and some IoT routers caught our eye[9]. Parts of our application armoring hygiene checks for the Linux ELF p_flags field in the program headers for the PT_GNU_STACK segment. If the executable bit is set and this header is present, the loader in modern Linux will map the segment as read-write-executable. However the MIPS binaries were odd as they completely lacked a PT_GNU_STACK segment. This results in the stack, heap, and other mappings being marked as read-write-execute. Thus a security weakness that was thought to have been addressed for over a decade appears to be present in these MIPS targets.

The fact that it has taken so long for this safety issue to come to light highlights the need for post-build safety checks on software. Other types of software testing are or are becoming industry standard, but it is not standard practice after compiling code to check that the resultant binaries have all intended safety features. Given how complex modern compiler toolchains are, edge cases and unintended results are inevitable. We hope that our work at CITL to bring issues like this one to light will provide the necessary motivation to introduce post-build testing to standard software production processes.

Technical Walkthrough

Readers who are not interested in the technical deep dive into the Linux kernel can skip to the Conclusion section.

[9] Our methodologies can be found online (https://cyber-itl.org/about/methodology/) and in the talks we have given at conferences such as BlackHat, DefCon, and CCC Congress.

Walkthrough of the Kernel

To understand this behavior we need to understand how Linux handles files without a stack segment. To do this we walk through the kernel to find the 'default' behavior and permissions applied. Starting in the `load_elf_binary()` function, we see the loader makes an architecture-specified call to determine if a READ flag implies EXECUTE flags:

https://github.com/torvalds/linux/blob/21b9f1c7e319f654de3b2574fe8d4e4114c9143f/fs/binfmt_elf.c#L876

```
if (elf_read_implies_exec(loc->elf_ex, executable_stack))
      current->personality |= READ_IMPLIES_EXEC;
```

Since this function uses the 'executable_stack' variable we work backwards to understand its definition.

The default value (`EXSTACK_DEFAULT`) is assigned to executable_stack at:
https://github.com/torvalds/linux/blob/21b9f1c7e319f654de3b2574fe8d4e4114c9143f/fs/binfmt_elf.c#L705

```
int executable_stack = EXSTACK_DEFAULT;
```

Later on the code checks the program headers for the PT_GNU_STACK and overwrites the default value:

https://github.com/torvalds/linux/blob/21b9f1c7e319f654de3b2574fe8d4e4114c9143f/fs/binfmt_elf.c#L810

```
case PT_GNU_STACK:
      if (elf_ppnt->p_flags & PF_X)
            executable_stack = EXSTACK_ENABLE_X;
      else
            executable_stack = EXSTACK_DISABLE_X;
      break;
```

The ELF files that we saw on these MIPS systems would never hit this switch case because they are missing the PT_GNU_STACK segment. This means the 'executable_stack' variable remains `EXSTACK_DEFAULT`.

Returning to the architecture specific call to '`elf_read_implies_exec`,' we see how the executable_stack variable is used.

https://github.com/torvalds/linux/blob/21b9f1c7e319f654de3b2574fe8d4e4114c9143f/arch/mips/kernel/elf.c#L329

```
int mips_elf_read_implies_exec(void *elf_ex, int exstack)
{
	if (exstack != EXSTACK_DISABLE_X) {
		/* The binary doesn't request a non-executable stack */
		return 1;
	}

	if (!cpu_has_rixi) {
		/* The CPU doesn't support non-executable memory */
		return 1;
	}

	return 0;
}
```

The `mips_elf_read_implies_exec()` checks the exstack argument to see if it is not `EXSTACK_DISABLE_X`, then falls through to return 1. This means that since the exstack is still `EXSTACK_DEFAULT`, the code will return 1. Thus, if the `PT_GNU_STACK` header is not present, it will always set the `READ_IMPLIES_EXEC` flag in 'current->personality'..

How does the `READ_IMPLIES_EXEC` flag affect the stack allocation during loading? The setup_arg_pages() function performs the actual mapping: https://github.com/torvalds/linux/blob/21b9f1c7e319f654de3b2574fe8d4e4114c9143f/fs/exec.c#L742

```
vm_flags = VM_STACK_FLAGS;

...

if (unlikely(executable_stack == EXSTACK_ENABLE_X))
	vm_flags |= VM_EXEC;
else if (executable_stack == EXSTACK_DISABLE_X)
	vm_flags &= ~VM_EXEC;
```

Here the code checks for `EXSTACK_ENABLE` and `EXSTACK_DISABLE`, but sets vm_flags to the default flag value first. Since `executable_stack` is still `EXSTACK_DEFAULT` the value of `VM_STACK_FLAGS` will decide the stack's execute flag, which is defined here: https://github.com/torvalds/linux/blob/21b9f1c7e319f654de3b2574fe8d4e4114c9143f/include/linux/mm.h#L277

```
#define VM_STACK_FLAGS   (VM_STACK | VM_STACK_DEFAULT_FLAGS | VM_ACCOUNT)
```

We see there is an addition of `VM_STACK_DEFAULT_FLAGS`, which is defined on a per architecture level. For the per-architecture value we once again look at the Linux implementation for MIPS as differing architectures can overwrite the value.

https://github.com/torvalds/linux/blob/21b9f1c7e319f654de3b2574fe8d4e4114c9143f/arch/mips/include/asm/page.h#L250

```
#define VM_DATA_DEFAULT_FLAGS \
      (VM_READ | VM_WRITE | \
       ((current->personality & READ_IMPLIES_EXEC) ? VM_EXEC : 0) | \
       VM_MAYREAD | VM_MAYWRITE | VM_MAYEXEC)
```

Here we see the `current->personality` flags are checked for `READ_IMPLIES_EXEC`, which was set previously and thus our default flags include `VM_EXEC`.

Implications of READ_IMPLIES_EXEC

The lack of a PT_GNU_STACK segment will not only affect how the stack's permissions are defined at load time but also other functions such as `mmap()`. Next let's follow the personality flags through the loader and kernel to get an idea of how this can impact a system.

Looking at `mmap` we see there are other locations this flag can cause `VM_EXEC` to be set on allocations:
https://github.com/torvalds/linux/blob/21b9f1c7e319f654de3b2574fe8d4e4114c9143f/mm/mmap.c#L1379

```
if ((prot & PROT_READ) && (current->personality & READ_IMPLIES_EXEC))
      if (!(file && path_noexec(&file->f_path)))
            prot |= PROT_EXEC;
```

Any mappings of anonymous or shared memory can be affected, causing the mappings to be marked executable.

Debian Mips32el on QEMU:

In order to verify expected behaviors we spun up a Debian 9.4 mips32el vm in QEMU. Below are the program headers and current mappings of `cat`:

```
user@debian:~$ readelf -l /bin/cat
```

```
Elf file type is DYN (Shared object file)
Entry point 0x2450
There are 11 program headers, starting at offset 52

Program Headers:
 Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
 PHDR           0x000034 0x00000034 0x00000034 0x00160 0x00160 R E 0x4
 INTERP         0x000194 0x00000194 0x00000194 0x0000d 0x0000d R   0x1
     [Requesting program interpreter: /lib/ld.so.1]
 ABIFLAGS       0x0001c8 0x000001c8 0x000001c8 0x00018 0x00018 R   0x8
 REGINFO        0x0001e0 0x000001e0 0x000001e0 0x00018 0x00018 R   0x4
 LOAD           0x000000 0x00000000 0x00000000 0x081b4 0x081b4 R E 0x10000
 LOAD           0x008f28 0x00018f28 0x00018f28 0x0039c 0x00528 RW  0x10000
 DYNAMIC        0x00021c 0x0000021c 0x0000021c 0x00120 0x00120 R   0x4
 NOTE           0x0001a4 0x000001a4 0x000001a4 0x00020 0x00020 R   0x4
 NOTE           0x0001f8 0x000001f8 0x000001f8 0x00024 0x00024 R   0x4
 GNU_RELRO      0x008f28 0x00018f28 0x00018f28 0x000d8 0x000d8 R   0x1
 NULL           0x000000 0x00000000 0x00000000 0x00000 0x00000     0x4

user@debian:~$ cat /proc/self/maps
559be000-559c7000 r-xp 00000000 08:01 20         /bin/cat
559d6000-559d7000 r-xp 00008000 08:01 20         /bin/cat
559d7000-559d8000 rwxp 00009000 08:01 20         /bin/cat
55a04000-55a25000 rwxp 00000000 00:00 0          [heap]
77985000-779a7000 rwxp 00000000 00:00 0
779a7000-77b42000 r-xp 00000000 08:01 261167     /usr/lib/locale/locale-archive
77b42000-77cae000 r-xp 00000000 08:01 130656     /lib/mipsel-linux-gnu/libc-2.24.so
77cae000-77cbe000 ---p 0016c000 08:01 130656     /lib/mipsel-linux-gnu/libc-2.24.so
77cbe000-77cc1000 r-xp 0016c000 08:01 130656     /lib/mipsel-linux-gnu/libc-2.24.so
77cc1000-77cc4000 rwxp 0016f000 08:01 130656     /lib/mipsel-linux-gnu/libc-2.24.so
77cc4000-77cc6000 rwxp 00000000 00:00 0
77cc6000-77ce9000 r-xp 00000000 08:01 130652     /lib/mipsel-linux-gnu/ld-2.24.so
77ced000-77cef000 rwxp 00000000 00:00 0
77cf3000-77cf4000 r-xp 0019a000 08:01 261167     /usr/lib/locale/locale-archive
77cf4000-77cf6000 rwxp 00000000 00:00 0
77cf6000-77cf7000 r--p 00000000 00:00 0          [vvar]
77cf7000-77cf8000 r-xp 00000000 00:00 0          [vdso]
77cf8000-77cf9000 r-xp 00022000 08:01 130652     /lib/mipsel-linux-gnu/ld-2.24.so
77cf9000-77cfa000 rwxp 00023000 08:01 130652     /lib/mipsel-linux-gnu/ld-2.24.so
7fae9000-7fb0a000 rwxp 00000000 00:00 0          [stack]
7ffff000-80000000 rwxp 00000000 00:00 0
```

```
user@debian:~$ uname -a
Linux debian 4.9.0-6-4kc-malta #1 Debian 4.9.82-1+deb9u3 (2018-03-02) mips GNU/Linux
```

Here we see the expected output given what was observed in the kernel with the stack, heap, and other mappings marked as executable.

Checking other programs shipped with Debian MIPS, in `/usr/bin` there were only 6 programs that included a PT_GNU_STACK segment, one of which was marked RWX. The other 5 were RW. The remaining 322 all lacked the PT_GNU_STACK segment and so, at runtime, would have RWX mappings similar to the /bin/cat given above.

As the kernel version (4.9) is after the non-executable stack patch was introduced yet the vast majority of binaries still lacked basic stack DEP, we observe the lag and interplay between compiler toolchains creating binaries that opt-in to the DEP now being supported by the kernel. This largely renders the kernel security feature meaningless in such systems.

For final verification, we wrote a small program to jump execution onto the stack then back to main. By placing a '`jr $ra`' instruction on the current stack address and calling it like a function pointer we were able to jump on and back off the stack. Doing this and single stepping allowed us to verify that the stack was indeed executable in run time.

We confirmed on other CPU architectures that the default behavior during runtime for a binary lacking a RW PT_GNU_STACK segment results in the stack and all anonymous mappings marked as RWX, thus empirically confirming the need for a RW PT_GNU_STACK segment to enable core DEP for applications within Linux.

Why is GCC/ld not emitting a PT_GNU_STACK segment?

Now we ask the question, why is GCC/ld not emitting a PT_GNU_STACK segment for Linux MIPS targets, even several years after the kernel could honor it? After digging through Linux and GCC mailing lists we found interesting answers.

The initial problem is as we have described: Because of differences in MIPS FPU coprocessors, sometimes the Linux kernel will need to emulate instructions for userland. Since there are non-fpu instructions that will be in the delay slot, these instructions need to run in the userland context, so they are placed on the userspace thread stack and executed from there. This is captured in the following 2014 comment by David Daney from Cavium:

https://patchwork.kernel.org/patch/5039371/

"*MIPS floating point support requires that any instruction that cannot be directly executed by the FPU, be emulated by the kernel. Part of this emulation involves executing non-FPU instructions that fall in the delay slots of FP branch instructions. Since the beginning of MIPS/Linux time, this has been done by placing the instructions on the userspace thread stack, and executing them there, as the instructions must be executed in the MM context of the thread receiving the Emulation.*

Because of this, the de facto MIPS Linux userspace ABI requires that the userspace thread have an executable stack. It is de facto, because it is not written anywhere that this must be the case, but it is never the less a requirement.

Problem:

How do we get MIPS Linux to use a non-executable stack in the face of the FPU emulation problem?

Since userspace desires to change the ABI, put some of the onus on the userspace code. Any userspace thread desiring a non-executable stack, must allocate a 4-byte aligned area at least 8 bytes long with that has read/write/execute permissions and pass the address of that area to the kernel with the new sys_set_fpuemul_xol_area system call.

This is similar to how we require userspace to notify the kernel of the value of the thread local pointer."

Another attempt at addressing the non-executable stack problem occurred in 2015 and can be found here: https://www.linux-mips.org/archives/linux-mips/2015-08/msg00118.html

Both attempts seem to be stalled for a number of reasons, mostly because they break userland and backwards compatibility. The authors of these patches acknowledged that the lack of a PT_GNU_STACK has the behavior of making the stack executable.

As noted in our timeline at the beginning of this document, a 2016 kernel patch, present in v4.8~rc1 and later, finally moved the FPU emulation off of the stack and into a newly mapped region.

https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=432c6bacbd0c16ec210c43da411ccc3855c4c010

This theoretically allows a compiler to include a PT_GNU_STACK segment without breaking userspace, but compilers have not all caught up to make use of that new feature. GCC/ld

(6.3.0), released at the end of 2016 and the primary compiler for many environments such as Debian MIPS, appear to default to not emitting a PT_GNU_STACK segment.

LLVM 6.0.0 defaults to emitting a PT_GNU_STACK with RW permissions *when cross compiling*, targeting mips-linux, however we found that Clang (3.8.1-24) running in a Debian MIPS environment *still* outputs a RWX stack segment for both MIPS 32 and 64.

Looking over the GCC/glibc mailing list, there was discussion and pre-work to attempt to have the compiler toolchain ready to support the non-executable PT_GNU_STACK that was being finalized in 2016.

https://sourceware.org/ml/binutils/2016-02/msg00087.html
https://sourceware.org/ml/libc-alpha/2016-02/msg00076.html

As mentioned above, the vast majority of the firmware images CITL has reviewed appear to have been built using toolchains that do not emit a RW PT_GNU_STACK even after the kernel patches had been released to permit this. We wondered about more recent versions of GCC. Going through the release change notes for GCC, https://www.gnu.org/software/gcc/releases.html, from 2016 through 2018, we find no reference to the non-executable stack changes for MIPS.

To reiterate the compiler defaults and capabilities mentioned above, 6.3.x is the version of GCC MIPS that is available through the main distro package managers for Ubuntu, Debian, …, for Intel/AMD and MIPS. This means that the vast majority of product manufacturers and developers will be using this toolchain. The GCC 8.2.0 MIPS compiler toolchain packages are available in unstable and experimental branches for our Linux systems. Even assuming a vendor builds their commercial products using unstable/experimental branches for their development environments, which we find unlikely, GCC 8.2 MIPS still defaults to an executable stack. It is, using 8.2, possible to specify a non-executable stack to the linker explicitly which then correctly outputs a GNU_STACK RW segment but we find very little evidence of this in deployed systems. Similarly, the stable package of GCC MIPS cross-compilers for Linux distributions is GCC 6.3.0. Thus for GCC compilers we note the following challenges: 1) the stable package version being distributed and used is 6.3, and 2) the unstable/experimental 8.2 MIPS packages default to an executable stack on MIPS binaries. These issues make the fact that GCC 8.2 *can* output the correct stack permissions relatively moot in practice[10].

[10] A GCC 8 cross compiler package is listed in the unstable branch for Debian with dependencies that did not allow even our unstable and experimental branch systems to correctly retrieve and install the package correctly. Across hundreds of current (as of September 2018) firmware images we note that binaries for home routers, unsurprisingly, have been built with toolchains comparable or earlier than 6.3.0. Further we noticed, as we mention below, that the same compiler toolchains we testing natively within a MIPS environment and being used to cross-compile to MIPS targets produced different PT_GNU_STACK outputs in the case of Clang.

Even assuming a toolchain that produced binaries with a default PT_GNU_STACK RW and a recent kernel that permits this without breaking floating point compatibility, there is a significant problem. Irrespective of the markings of the stack, heap, and anonymous mappings, the kernel NX stack patch we have been discussing introduces a new segment in all processes at a ***fixed location*** and that is marked as readable, writable, and ***executable*** (*RW**X***).

We now take a closer look at this newly introduced issue.

A closer look at the NX stack Linux kernel patch and the introduction of an ASLR/DEP bypass primitive

Even though it was becoming apparent that it was not only rare to find binaries built with the safety of a non-executable stack in shipping products we wanted to evaluate the runtime of binaries that embodied a RW GNU_STACK segment. While we expected things to be fine, we were surprised by what we observed when running on a modern MIPS kernel. We used clang, in cross compile mode, in order to build a few test programs with correct RW stack segments, as we noted before that we were unable to get native clang output a proper RW GNU_STACK. We then ran the binary to inspect how the segments were mapped. While the stack and other segments are now correctly marked RW, we found that the patch introduced in 4.8-rc1 (commit: 432c6bacbd0c16ec210c43da411ccc3855c4c010) created an RWX segment mapped at a static address for every userland process. A static address combined with a RWX permission effectively creates a universal ASLR and DEP bypass in every userland process.

```
00400000-00407000 r-xp 00000000 08:01 209        /home/user/test_stack
00416000-00417000 rwxp 00006000 08:01 209        /home/user/test_stack
77b0d000-77b0e000 r--p 00000000 00:00 0          [vvar]
77b0e000-77b0f000 r-xp 00000000 00:00 0          [vdso]
7fae4000-7fb05000 rw-p 00000000 00:00 0          [stack]
7ffff000-80000000 rwxp 00000000 00:00 0
```

The program has a non-executable stack as expected but that final mapping now shows a different RWX segment as can be seen above, highlighted in red. We compiled the binary for position independent code, to ensure an ASLR compatible binary. After running the program multiple times, rebooting and running it and other programs multiple times, the 0x7ffff000 mapping remained stable at this fixed address.

The following code demonstrates that we can indeed place code within the new at-risk segment, jump to the newly placed code, and execute it. This is essentially the same code that we used to test execution on the stack, but placed the ‘ja $ra’ onto the new fixed address mapping.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

int main(void) {
    // set a pointer to the vfpu emulation page address
    void* p = (void *)0x7ffff000;
    printf("%p\n", (void*)p);

    // construct a function pointer from p
    void (*func_ptr)(void) = p;

    // 'jr $ra' mips32el instruction bytes
    char code[] = {0x08, 0x00, 0xe0, 0x03, 0x00, 0x00, 0x00, 0x00};

    // copy the instruction to the vfpu page
    memcpy(p, code, 8);

    // call the function pointer, this should then directly return back
    (*func_ptr)();

    // print out the current maps of the process
    char cmd[200];
    sprintf(cmd, "cat /proc/%d/maps", getpid());
    system(cmd);

    return 0;
}
```

Output mipsel Debian 9.4

```
$ /test-exec
0x7ffff000
00400000-00407000 r-xp 00000000 08:01 239        /test-exec
00417000-00418000 rwxp 00007000 08:01 239        /test-exec
77a93000-77a94000 r--p 00000000 00:00 0          [vvar]
77a94000-77a95000 r-xp 00000000 00:00 0          [vdso]
7f8b3000-7f8d4000 rw-p 00000000 00:00 0          [stack]
7ffff000-80000000 rwxp 00000000 00:00 0
```

Taking a closer look at the patch we find where the mapping used for emulation is made: https://github.com/torvalds/linux/blob/21b9f1c7e319f654de3b2574fe8d4e4114c9143f/arch/mips/kernel/vdso.c#L110

```
base = mmap_region(NULL, STACK_TOP, PAGE_SIZE,
                   VM_READ|VM_WRITE|VM_EXEC|
                   VM_MAYREAD|VM_MAYWRITE|VM_MAYEXEC,
                   0, NULL);
```

The address of the mapping is set to STACK_TOP, which is defined:

https://github.com/torvalds/linux/blob/21b9f1c7e319f654de3b2574fe8d4e4114c9143f/arch/mips/include/asm/processor.h#L87

```
#define STACK_TOP ((TASK_SIZE & PAGE_MASK) - PAGE_SIZE)
```

Because TASK_SIZE is hardcoded for either 32 or 64 bit, and PAGE_MASK / PAGE_SIZE are both static values based on the MMU page size, the mapping is deterministic. This means the FPU emulation mapping segment is never randomized by ASLR. This is particularly problematic because it means that every userland process running on a Linux MIPS target has a static mapping that is RWX. To be more blunt: this provides a deterministic location for an exploit developer targeting Linux MIPS to place shellcode, providing a basic bypass ASLR and DEP primitive.

We also confirmed the same behavior occurs in a Debian 9.4 MIPS64 VM:

```
ffffcdc000-ffffcfd000  rwxp 00000000 00:00 0    [stack]
fffffff000-10000000000 rwxp 00000000 00:00 0
```

The mapping is at a slightly different location, but still with RWX permissions and a deterministic static address. If you find that your system has the mapping but is at a different address, it is possible your system is using a different page size.

Conclusion

We observed many Linux MIPS systems in the wild that are impacted by this. Further, the 2016 patches appear to be only partially effective at addressing the DEP issue, as they introduce a statically located mapping that is marked RWX. This means that the problem has been moved, rather than fixed.

From our analysis of specific Linux MIPS firmware images we notice an astonishing lack of many basic security features in addition to those mentioned in this paper, and also a blend of programs exhibiting this specific problematic behavior and a few that do not -- even on the same system. One hypothesis for this is that vendors may be acquiring different subsections of code from various subcontractors or 3rd parties and/or the different output based on whether native

or cross compilation is being performed. We examine these products and their measurements further in our recent report on the home router market[11].

Irrespective of the cause of the variance, we note that the majority of Linux MIPS builds are likely to contain these issues. The recency of the FPU emulation patch, and the means it introduces to sidestep ASLR and DEP cause significant concern.

Many of these concerns could be detected before releasing software by companies in their products by implementing simple tests and checks to ensure the resultant binaries have their intended safety features. Our evaluation of Linux MIPS products shows this is not happening.

Historical use of an executable stack on other architectures lingered for years but has largely been addressed in modern (x86) Operating Systems. Linux MIPS suffered from this for many years, and continues to do so. The recent attempt an enabling mitigations further introduces new security concerns.

We believe Linux MIPS will continue to be a soft target for many years to come.

Acknowledgements

Parker Thompson was the lead researcher and wrote the first draft of this report. Mudge performed additional research and led the writing and structure of the final report.

Patrick Stach provided invaluable contributions in determining the cause and impact of the Linux/MIPS issues discussed herein.

The authors wish to thank Sarah Zatko and Tim Carstens for their significant assistance in writing and editing portions of this report.

[11] https://cyber-itl.org/assets/papers/2018/build_safety_of_software_in_28_popular_home_routers.pdf