A return-oriented programming defense from OpenBSD

By **Jonathan Corbet**
August 30, 2017

Stack-smashing attacks have a long history; they featured, for example, as a core part of the Morris worm back in 1988. Restrictions on executing code on the stack have, to a great extent, put an end to such simple attacks, but that does not mean that stack-smashing attacks are no longer a threat. Return-oriented programming (ROP) has become a common technique for compromising systems via a stack-smashing vulnerability. There are various schemes out there for defeating ROP attacks, but a mechanism called "RETGUARD" that is being implemented in OpenBSD is notable for its relative simplicity.

In a classic stack-smashing attack, the attack code would be written directly to the stack and executed there. Most modern systems do not allow execution of on-stack code, though, so this kind of attack will be ineffective. The stack *does* affect code execution, though, in that the call chain is stored there; when a function executes a "return" instruction, the address to return to is taken from the stack. An attacker who can overwrite the stack can, thus, force a function to "return" to an arbitrary location.

That alone can be enough to carry out some types of attacks, but ROP adds another level of sophistication. A search through a body of binary code will turn up a great many short sequences of instructions ending in a return instruction. These sequences are termed "gadgets"; a large program contains enough gadgets to carry out almost any desired task — if they can be strung together into a chain. ROP works by locating these gadgets, then building a series of stack frames so that each gadget "returns" to the next.

This technique allows the construction of arbitrary programs on the stack without the need for execute permission on the stack itself. It is worth noting that, on a complex-instruction-set architecture like x86, unexpected gadgets can be created by jumping into the middle of a multi-byte instruction, a phenomenon termed "polymorphism". Needless to say, there are tools out there that can be used by an attacker to locate gadgets and string them together into programs.

The RETGUARD mechanism, posted by Theo de Raadt on August 19, makes use of a simple return-address transformation to disrupt ROP chains and prevent them from executing as intended. It takes the form of a patch to the LLVM compiler adding a new `-fret-protector` flag. When code is compiled with that flag, two things happen:

- The prologue to each function (the code that runs before the body of the function itself) exclusive-ORs the return address on the stack with the value of the stack pointer itself.
- The epilogue, run just before the function returns, repeats the operation to restore the return address to its initial value.

The exclusive-OR operation changes the return address into something that is effectively random, especially when address-space layout randomization is used to place the stack at an unpredictable location. With this change, the first gadget used by a ROP sequence will, when it attempts the second step above, transform the return address into something unpredictable and, most likely, useless to an attacker. That will stop the chain and thwart the attack.

There is, of course, a significant limitation here: a ROP chain made up of exclusively polymorphic gadgets will still work, since those gadgets were not (intentionally) created by the compiler and do not contain the return-address-mangling code. De Raadt acknowledged this limitation, but said: "we believe once standard-RET is solved those concerns become easier to address separately in the future. In any case a substantial reduction of gadgets is powerful".

Using the compiler to insert the hardening code greatly eases the task of applying RETGUARD to both the OpenBSD kernel and its user-space code. At least, that is true for code written in a high-level language. Any code written in assembly must be changed by hand, though, which is a fair amount of work. De Raadt and company have done that work; he reports that: "We are at the point where userland and base are fully working without regressions, and the remaining impacts are in a few larger ports which directly access the return address (for a variety of reasons)". It can be expected that, once these final issues are dealt with, OpenBSD will ship with this hardening enabled.

It makes sense to ask whether this relatively straightforward hardening technique could be applied to the Linux kernel as well. Using LLVM to build the kernel is not yet a viable option, but it should be possible to reimplement the RETGUARD transformations as a GCC plugin module. The tiresome task of fixing up the assembly code would also need to be done; the `objtool` utility could probably be pressed into service to help with this task. But the patch that emerged would not be small.

If any benchmarks have been run to determine the cost of using RETGUARD, they have not been publicly posted. The extra code will make the kernel a little bigger, and the extra overhead on every function is likely to add up in the end. But if this technique can make the kernel that much harder to exploit, it may well justify the extra execution overhead that it brings with it. All that's needed is somebody to actually do the work and try it out.

(Log in to post comments)

A return-oriented programming defense from OpenBSD
Posted Aug 30, 2017 2:25 UTC (Wed) by **pabs** (subscriber, #43278) [Link]

That sounds similar to but not the same as the equivalent protection in RAP from grsecurity, I wonder if the OpenBSD implementation was influenced by grsecurity's patent on RAP.

https://www.grsecurity.net/rap_announce.php
https://www.grsecurity.net/rap_faq.php

A return-oriented programming defense from OpenBSD
Posted Aug 30, 2017 3:26 UTC (Wed) by **smurf** (subscriber, #17840) [Link]

Probably not; RAP is considerably more involved (and causes a lot more slowdown).
Using the stack pointer itself to encode the return address is an interesting idea that should help thwart such attacks with significantly lower overhead.

A return-oriented programming defense from OpenBSD
Posted Aug 30, 2017 6:54 UTC (Wed) by **PaXTeam** (guest, #24616) [Link]

it has nothing to do with RAP and it's not quite that novel (and secure) either:
https://twitter.com/grsecurity/status/899294869105106944 and https://pax.grsecurity.net/docs/pax-future.txt .

A return-oriented programming defense from OpenBSD
Posted Aug 30, 2017 9:04 UTC (Wed) by **alonz** (subscriber, #815) [Link]

Note that the scheme used in RAP has a different trade-off - RAP needs two extra registers (which probably hurts performance) but it also doesn't alter the return address iterator. Modifying the return address are runtime, the way this technique does, will invalidate the processor's branch prediction and hurt performance.

A return-oriented programming defense from OpenBSD
Posted Aug 30, 2017 11:21 UTC (Wed) by **roc** (subscriber, #30627) [Link]

I don't think this would interfere with the return stack buffer. Probably the RSB just predicts the return address using an internal stack and verifies that the return address, when popped, matches its prediction. It doesn't matter whether the memory containing the return address was mangled temporarily.

A return-oriented programming defense from OpenBSD
Posted Aug 30, 2017 4:18 UTC (Wed) by **luto** (subscriber, #39314) [Link]

I wonder how easy this is to defeat if the stack address is already known.

A return-oriented programming defense from OpenBSD
Posted Aug 30, 2017 5:06 UTC (Wed) by **josh** (subscriber, #17465) [Link]

Or if the stack address is reasonably predictable.

A return-oriented programming defense from OpenBSD
Posted Aug 30, 2017 13:58 UTC (Wed) by **epa** (subscriber, #39769) [Link]

In most cases a function is called from only a finite number of places. If the number of such places is less than 256, then instead of storing a return address on the stack you could store a byte. At the end of the function the compiler generates code to jump to one of several possible addresses depending on the value of that byte. Then no matter what the attacker overwrites he can't jump to a gadget of his choosing, only to a small number of points that call the function in normal use anyway. And if there are only 5 possible return points but the byte is overwritten with 99, that's a clean crash ('stack corruption detected') rather than a jump to some random address. The stack becomes smaller too. (Performance would suffer if there are loads of possible return points and the compiler generates an endless if-then-goto-else chain, but you could set an upper limit or disable the mechanism for performance-critical code, which can then be audited more carefully than usual.)

Here I am thinking of the program as a single lump of object code which is compiled and linked in its entirety before running. Obviously if you have loadable modules or you are writing a dynamically linked library you have to allow more flexibility in return addresses.

A return-oriented programming defense from OpenBSD
Posted Aug 31, 2017 0:12 UTC (Thu) by **droundy** (subscriber, #4559) [Link]

Nice idea! Presumably you'd want this to use a simple lookup table (with a bounds check), in which case the performance wouldn't be particularly affected by a large number of callers.

A return-oriented programming defense from OpenBSD
Posted Aug 31, 2017 8:34 UTC (Thu) by **epa** (subscriber, #39769) [Link]

Yes, a lookup table might perform better for large numbers of callers, while if there are just two possible values a couple of inline test-and-jumps would likely be faster. (And if there are a hundred possible callers but one of them is way more frequent than the others, it might still be faster to test for that one first.) It depends on benchmarks and would likely vary by platform and even by different CPUs in the same platform.

If the program is not multithreaded, and static analysis shows that a function is not re-entrant, then its return address (or a number to look up the return address) does not need to be on the stack at all. It can be at a fixed location, making it harder for an attacker to overwrite, and saving some stack space too.

Indeed, there's a case for saying that all re-entrant functions should need to be explicitly tagged as such by the programmer, since they need more care in writing. If the function isn't tagged as re-entrant, and the compiler cannot statically prove that it can never be called by some chain starting from itself, then that's a compile-time warning or error. On the other hand, if a function is never re-entrant, all its local variables can be allocated statically away from the stack. Who knows, perhaps some optimizing compilers do this already.

A return-oriented programming defense from OpenBSD
Posted Aug 31, 2017 9:31 UTC (Thu) by **karkhaz** (subscriber, #99844) [Link]

This is a cool idea, but I'm confused about two points.

1. You say
> and the compiler cannot statically prove that it can never be called by some chain starting from itself

AFAIK there will be plenty of situations where a function cannot be called through a chain starting from itself, but a static analysis cannot prove this, so the compiler will emit false positives (i.e. tell you that you need to tag the function when you should not). This is due to function pointers; static analyses typically over-approximate what concrete values function pointers might have at runtime. Analyses that have a more precise idea about function pointer addresses are typically very slow.

Note also that the analysis would need to have the entire program at its disposal to determine the values of function pointers, while here we're talking about the compiler (which only has access to a single translation unit). There are tools like CBMC [1] that don't have much trouble with analyzing entire programs and serve as a drop-in replacement to GCC, and there has been some work [2] to the Clang Static Analyzer that would enable it to analyze the entire program at once (still in discussion), but these are both way beyond the capabilities of a regular compiler anyway. Alternatively, it could be done as a link-time optimization (well, it would be a link-time analysis, but the line between analysis and optimization is quite fine), as at link-time you have the whole program, though I don't know what the exact capabilities of LLVM and GCC's LTO are and whether this would be possible.

2. I'm not sure what reentrancy has to do with this, would you mind elaborating?

[1] http://www.cprover.org/cbmc/
[2] https://reviews.llvm.org/D30691

A return-oriented programming defense from OpenBSD
Posted Aug 31, 2017 13:41 UTC (Thu) by **epa** (subscriber, #39769) [Link]

That's right, static analysis won't be able to show all cases where a function isn't re-entrant. The programmer would have to annotate some functions with a tag to quieten the warning and would presumably add a comment with a human-readable explanation (or proof) of why it's OK. Yes, analysis of the whole program as a unit is necessary -- don't compilers have whole-program optimization modes nowadays?

After my first paragraph I went off on a separate idea which was that the stack could be avoided altogether if a function can only be executing 'once' -- it cannot call itself so it cannot appear twice in a call stack. In that case you can set aside a static area of memory, which is read-write of course, but is physically separate from the stack and so perhaps less likely to be trampled in a typical stack smashing attack. This might not really be a win, if it just means that memory trampling attacks against this static area become easier.

A return-oriented programming defense from OpenBSD
Posted Aug 31, 2017 10:16 UTC (Thu) by **ibukanov** (subscriber, #3942) [Link]

This suggests to go back to early Fortran model with no stack but static locations for return addresses unless the function is recursive.

A return-oriented programming defense from OpenBSD
Posted Aug 31, 2017 15:26 UTC (Thu) by **mathstuf** (subscriber, #69389) [Link]

Wouldn't this break the ABI and `dlsym` looking up and using that function? Or would this approach only be viable for static functions or functions going into executables?

A return-oriented programming defense from OpenBSD
Posted Aug 31, 2017 19:15 UTC (Thu) by **nix** (subscriber, #2304) [Link]

It would only be viable for static functions whose addresses are not leaked (whether dlsym() counts as such a leak is questionable). Simply taking the function's address is probably enough to invalidate it, particularly given the existence of things like register_printf_function(), or, heck, atexit().

A return-oriented programming defense from OpenBSD
Posted Aug 30, 2017 20:15 UTC (Wed) by **michaeljt** (subscriber, #39183) [Link]

I wondered recently about how useful copying the return address to a local variable at the bottom (address-wise) of the frame and comparing them before returning would be as a defence technique.

A return-oriented programming defense from OpenBSD
Posted Aug 31, 2017 12:00 UTC (Thu) by **sorokin** (subscriber, #88478) [Link]

This still allows overwriting of the return address of the caller of the current function.

A return-oriented programming defense from OpenBSD
Posted Aug 31, 2017 12:07 UTC (Thu) by **michaeljt** (subscriber, #39183) [Link]

> This still allows overwriting of the return address of the caller of the current function.

Yes, I realise that. It is at least harder though as you have to avoid overwriting your own in the process.

A return-oriented programming defense from OpenBSD
Posted Aug 31, 2017 12:04 UTC (Thu) by **sorokin** (subscriber, #88478) [Link]

Lots of different solutions were proposed in comments.

1. Xoring the return address stored in stack. (article)
2. Allowing a function to return only to a finite number of places using an array of possible return addresses. (epa)
3. Keeping return address in a global variable in the case the function is non-recursive. (epa)
4. Duplicate a return address at bottom of stack frame. (michaeljt)

Just for completeness I would like to say that another possible solution would be having two separate stacks. One for return addresses and for variables whose address is not taken. Another for variables whose address is taken.

A return-oriented programming defense from OpenBSD
Posted Aug 31, 2017 15:24 UTC (Thu) by **mathstuf** (subscriber, #69389) [Link]

There's also the way the Mill CPU is doing it where the CPU manages the call stack pointers for you and there's no access to them (except presumably through the debugger APIs, but I assume even that is read-only). Kind of like the split stack, but instead, it is just the way the CPU works.

A return-oriented programming defense from OpenBSD
Posted Aug 31, 2017 19:17 UTC (Thu) by **nix** (subscriber, #2304) [Link]

Intel has a similar thing they're calling 'shadow stacks'.

A return-oriented programming defense from OpenBSD
Posted Aug 31, 2017 18:07 UTC (Thu) by **wahern** (subscriber, #37304) [Link]

The dual-stack approach was merged into Clang.

http://dslab.epfl.ch/proj/cpi/
http://clang.llvm.org/docs/SafeStack.html

Though IIUC there are still some unresolved integration and interoperability issues involving threading, dynamic memory, etc, that make SafeStack undesirable to use long-term until those issues are resolved.

A return-oriented programming defense from OpenBSD
Posted Aug 31, 2017 22:18 UTC (Thu) by **thestinger** (subscriber, #91827) [Link]

The Clang implementation works well when paired with proper integration in libc, but glibc doesn't have that.

A return-oriented programming defense from OpenBSD
Posted Sep 11, 2017 13:29 UTC (Mon) by **itvirta** (guest, #49997) [Link]

> The dual-stack approach was merged into Clang.

So, if I got that right, they make another software controlled stack for variables/arrays that get their pointers taken, and keep the rest in the usual hardware stack?

I've sometimes wondered why CPUs don't implement a separate, protected, stack for the CALL/RETURN instructions.
That should deal with all sorts of overwriting and adding return pointers, if the stack was made read-only or completely
inaccessible to other instructions. But I don't know if there would be some prohibitive cost to that.

A return-oriented programming defense from OpenBSD
Posted Sep 11, 2017 18:40 UTC (Mon) by **wahern** (subscriber, #37304) [Link]

From Table 1 of the Code-Pointer Integrity (2014) paper:

```
                   Safe Stack  CPS    CPI
------------------------------------------
Average (C/C++)  |       0.0%   1.9%   8.4%
Median (C/C++)   |       0.0%   0.4%   0.4%
Maximum (C/C++)  |       4.1%  17.2%  44.2%
------------------------------------------
Average (C only) |      -0.4%   1.2%   2.9%
Median (C only)  |      -0.3%   0.5%   0.7%
Maximum (C only) |       4.1%  13.3%  16.3%
------------------------------------------
Table 1: Summary of SPEC CPU2006
performance overheads.
```

Safe Stack is the dual-stack mechanism. CPS(weak) and CPI (strong) are for dealing with function pointers in heap data.

A return-oriented programming defense from OpenBSD
Posted Sep 12, 2017 16:20 UTC (Tue) by **zlynx** (subscriber, #2285) [Link]

I believe that *every* recent CPU instruction set does a separate return stack. Well, as far as I can tell RISC-V puts it in a register, then it is up to a function caller to save the "ra" register wherever it wants before making the call. It can save it to the stack, a different stack or a linked list, the processor doesn't care. Itanium was similar, return addresses were saved in registers, and the registers would overflow into a separate stack.

If only people would stop relying on the x86 / amd64 ISA.

A return-oriented programming defense from OpenBSD
Posted Sep 12, 2017 18:32 UTC (Tue) by **excors** (subscriber, #95769) [Link]

RISC-V sounds similar to ARMv8 AArch64, where (if I understand correctly) the "BLR" instruction branches and stores the return address in the X30 (LR) register, and the "RET Xn" instruction returns to the address stored in some register, and that's the only proper way to call a function. A non-leaf function can preserve X30 however it wants; usually it will push/pop X30 on the stack associated with the slightly magic SP register, but the push/pop instructions (store-and-decrement/load-and-increment) can use any register as the index, so I think you could create a separate control stack for approximately zero cost (just one more reserved register) to keep the frame pointers and return addresses away from all the "char surely_this_is_big_enough[256];" buffers and other local variables.

(ARMv7/AArch32 is similar but more confusing, because there are lots of mostly-deprecated ways of returning by using PC as a destination register, and you usually push/pop LR/PC in the same instruction as all the other registers you want to preserve (whereas AArch64 can only push/pop a pair of registers at once), and the Thumb instruction encoding has lots of limitations, and there are only half as many registers as AArch64, so a separate control stack might be significantly more expensive there.)

A return-oriented programming defense from OpenBSD
Posted Aug 31, 2017 23:04 UTC (Thu) by **shemminger** (subscriber, #5739) [Link]

Windows also has a similar (and somewhat richer) set of ROP defenses:
https://msdn.microsoft.com/en-us/library/windows/desktop/...(v=vs.85).aspx
Though there are IP issues in using some of this in Linux.

Also, there is some evidence that these are reducing the use of ROP in exploits:
https://www.endgame.com/blog/technical-blog/rop-dying-and...