Using function expansion to increase Shadow Stack viability

Pedro Terra Delboni, Heitor Boschirolli, João Moreira, Sandro Rigo

[1]Instituto de Computação – Universidade Estadual de Campinas (UNICAMP)

Abstract. *Refined protections against Control-Flow Hijack involve the use of a shadow stack. Unfortunately, their implementation induces an increase in execution time which in many cases is unacceptable. A proposed solution to this issue is to expand selected calls, but this solution hasn't been tested yet. In this paper, we'll explain our strategy to evaluate this proposition.*

Resumo. *Proteções refinadas contra Sequestro de Controle de Fluxo involvem o uso de uma Shadow Stack. Infelizmente suas implementações induzem um aumento no tempo de execução que em vários casos é inaceitável. Uma solução proposta para esse problema é expandir chamadas de funções em lugares chaves do programa. Nesse artigo vamos explicar a nossa estratégia para avaliar essa proposta.*

1. A brief story of Control-Flow Hijack

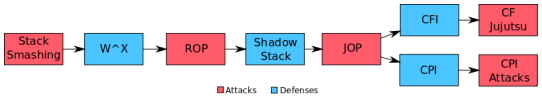

Figure 1. Iterations of attacks and defenses.

Figure 1 shows a brief sequence of attacks and defenses that illustrate the importance of a shadow stack. Stack Smashing [One 1996] presented the dangers of unprotected stacks by injecting code and replacing the stack's return address to a pointer to the injection. Write XOR Execute is a hardware defense that doesn't allow execution of instructions at writable memory. ROP [H. 2007] is an attack that instead of injecting code, manipulates return addresses to execute code already in the program in an unpredicted order which gives the attacker control of the machine. Shadow Stacks [Cowan et al. 1998] is a defense against stack corruption that involves creating a second, more protected, stack which will contain redundancy of return addresses so they can be validated before exiting a function. JOP [S. et al. 2010] is another attack that uses function pointers instead of return address. This lead to defenses like CFI [Abadi et al. 2005] and CPI [Kuznetsov et al. 2014], which if not implemented strictly can still be bypassed by other attacks [I. et al. 2015] [Evans et al. 2015], and a good implementation needs a shadow stack.

2. Expanding Functions (inline)

By expanding (inlining) a function, no call is issued, thus no return address can be corrupted. Return addresses are also the most frequently accessed code pointers inside our program, so by removing them we are removing most of the places which need to be protected, removing the cost to protect them.

2.1. Selecting functions to inline

Inlining every function is not an option, because it would make the program binary a lot bigger. Another issue is that by inlining functions, we may be removing the cost of protecting a call, but we may also be adding the cost of cache misses, since two different calls to the same function will now lead to two different parts of the binary.

In order to maximize the efficiency, we need to inline the calls which are used the most. This means that our choice will be based on a specific execution flow. In order to choose these calls, we modified the compiler to create one global variable associated with each call, and added instructions right before the call to increment this variable. At the end of our desired execution, we have a list of how many times each call was made. This will help us determine which calls to inline.

2.2. Inlining a call

Compilers already have the capacity to inline a call, but they can't inline **any** call. In order for a call to be expanded, the callee must either be at the same file that the caller or the expansion must be made at a link-time optimizer. Unfortunately, link-time optimizers aren't well adopted yet, and many projects can't be compiled with them.

We propose two solutions:

- **Cloning calls** The simplest solution to remove the call and return instructions was to instead of expanding the function we modify the compiler to create a clone of the callee and replacing the call and return instructions to jumps in and out of the clone.
- **Inlining calls** A more complex solution is to use the compiler to get every function that will be inlined and create a special module that will contain only a copy of the function with every symbol used by it marked as an external one. We'll fuse this module with the one with the call and point the call to the new function, so the compiler will now be able to inline it.

3. Evaluating the results

We are finishing the implementation of both solutions and once done we'll try to answer the following questions:

- Given a specific execution flow is it possible to inline a set of calls that will make the cost of shadow stacks negligible?
- Is there a set of functions that will make so that most execution flows won't be penalized by the cost of the shadow stack?
- Is the impact of cash misses on expanded functions greater than the impact of the shadow stack, and if not, is this impact acceptable?

References

Abadi, M., Budiu, M., Erlingsson, Ã., and Ligatti, J. (2005). Control-flow integrity: Principles, implementations, and applications. *ACM SIGSAC Conference on Computer and Communications Security (CSS)*.

Cowan, C., Pu, C., Maier, D., Hintony, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., , and Zhang, Q. (1998). Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. *8st USENIX Security Symposium*.

Evans, I., Fingeret, S., Gonzalez, J., Otgonbaatar, U., Tang, T., Shrobe, H., Sidiroglou-Douskos, S., Rinard, M., and Okhravi, H. (2015). Missing the point(er): On the effectiveness of code pointer integrity. *IEEE Symposium on Security and Privacy*.

H., S. (2007). The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). *Proceeding CSS 07 Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561.

I., E., F., L., U., O., H., S., M., R., H., O., and S, S.-D. (2015). Control jujutsu. *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS '15*, pages 901–913.

Kuznetsov, V., Szekeres, L., Payer, M., Candea, G., Sekar, R., , and Song, D. (2014). Code-pointer integrity. *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

One, A. (1996). Smashing the stack for fun and profit. *Phrack Magazine 49(14)*.

S., C., L., D., A., D., A., S., H., S., and M., W. (2010). Return-oriented programming without returns. *7th ACM conference on Computer and communications security - CCS Í0*.