MSpec: Automated Safety Reasoning for OS Compartmentalization

Vlad-Andrei Bădoiu
University Politehnica of Bucharest
Bucharest, Romania

Abstract

An increasingly vast array of security mechanisms that can be used to improve the safety of software including address-space isolation, CPU protection levels, memory-protection keys, TEEs, capabilities, as well as software variants including control-flow and data-flow integrity, address-sanitization and so forth. How should we use these mechanisms, and in which combination, to ensure that our software is safe and achieves the best possible performance? The status quo chooses safety mechanisms manually, at design time, but this sub-optimal in many ways. In this thesis our aim is to automatically decide which mechanisms should be used to make such an operating system safe. We propose MSpec, a domain-specific language and a tool that performs automatic safety reasoning and present preliminary results.

1 Introduction

The operating systems (OS) security scene is rapidly evolving. We have many security mechanisms available on an average OS, ranging from hardware mechanisms (e.g. MPK [13], CHERI [14], EPT) to software-based (e.g CPI [5], CFI [1], SFI, ASAN [11]). Mechanisms can be used to create compartments and enforce different properties such as read/write/execute policies. Given a set of components, how should we apply these security mechanisms to them? Which OS components should be placed in different MPK compartments, which should be hardened using ASAN, and so forth? Further, flexible isolation [7] has shown us that the configuration space resulting from mixing mechanisms and creating compartments (e.g. VMs in the case of EPT) grows exponentially, with hundreds of configurations(mapping of security mechanisms to components) available for only a few OS components and mechanisms. However, reasoning about the resulting safety properties of a given compartmentalization strategy is currently done manually and is both tedious and error prone. [6]

In this PhD thesis we propose an automated approach to safety reasoning based on a new Domain Specific Language (DSL) that models the behaviour of OS components and the interaction between them while taking into account which safety mechanisms are used. Each component is described by a set of safety properties—which explain effects of running this component may have on other components—and a set of requirements that it expects others to conform with. Each component comes with a set of safety properties which is dictated by the language it is implemented in and its complexity; for instance, a component written in C and accessing memory via pointers may, in the worst case, write to any memory address. A safety property allow us to express this potential behaviour and, in conjunction with the safety requirements of other components, we can automatically decide if a given configuration is safe or not. When a security mechanism is applied to a component, it modifies (strengthens) its safety properties. In MSpec we model the application of security mechanisms to a component as a transformation which alters the properties of the component they apply to. Thus, by using a set of available transformations we model the effects of applying various security mechanisms to components, with the end goal of finding configurations that satisfy all the constraints required by each component. Configurations are partially ordered, all deriving configurations of a solution are solutions but with a inferior performance. To be able to compare any two configurations we introduce a cost function that estimates the overhead of a configuration based on the used transformations.

To showcase the applicability of our work we prototyped a tool called MSpec that can be easily integrated into the build system of an OS to automatically generate configurations given the components, transformations and strategies. We integrated MSpec with FlexOS [6, 7] due to the high configuration spaced enabled by its flexible isolation approach.

MSpec differs from other compartmentalization solutions through generality. Other solutions only model the isolation property of some mechanisms and use it only for selected use cases such as least privilege[9, 10].

2 Overview

In Figure 1 we present an overview of MSpec. The core of MSpec is a domain-specific language that enables the specification of component properties and requirements in a fine- or a coarse-grained manner. Fine-grained properties can used to specify byte-level memory properties, whereas coarse-grained refers to component-level overall properties. Currently MSpec properties are specified in the context of a specific compartment. For components defined at function level we target MSpec based code annotations. Kernel and application components are either described in MSpec by the corresponding developers or a default specification is provided based on the programming language (e.g. memory safety in safe Rust code) or even produced automatically by the compiler or an analysis tool. For instance, a verified cooperative scheduler written in Dafny [8], has the following coarse-grained MSpec specification:

```
[Memory Access] ::= R, W
[Call] ::= X
[API] ::=  (thread_add, SYMB) (thread_remove, SYMB)\
          (schedcoop_yield, SYMB)
[Requires] ::= X
```

The component properties specified are that (1) it requires read/write access over shared memory, data that the scheduler is supposed to access and is shared via pointers; (2) it will execute according to the expected control flow, guaranteed by Dafny; (3) it exports an API consisting of three symbols to outer components; and (4) the only capability provided to outer components is the execution of current component code via the exposed symbols.

Transformation specification . Similarly, we model how a security mechanism affects the safety of a component using transformations. A transformation can also specified in MSpec and captures how component properties change as well as the way in which mechanisms create compartments and the relation between them.

Given a set of available transformations and the properties and requirements of each component, MSpec is able to automatically find configurations that satisfy all the constraints. The user can also specify additional constraints not captured by the component safety requirements (e.g. do not use transformation X alongside Y - shown as (C) in Fig. 1).

To reason about the safety of a configuration, MSpec reduces the problem to an adapted graph colouring problem where each colour represents a compartment and it's attached set of transformations. In our representation, nodes represent components whereas edges are interaction between components. Nodes can have multiple colours attached to them because we can have nested compartments (e.g. an MPK compartment in an EPT compartment). A solution is a colour configuration where all nodes with the same colour are compatible. We also target overall compatibility between colors. (e.g. the least permissive properties from container X are compatible with the most constrained requirements from container Y) Since there are typically many safe solutions, we want to find the one with the best expected performance. To this end, we introduce a cost function (e.g. number of compartments used) that orders configurations based on their expected performance (D in Fig. 1).

The way we specify the safety properties of a component can help us solve different problems using MSpec. The default behaviour of MSpec is the safety oriented strategy, where we use the properties of the language a component is written in to define its safety properties (e.g. code written in Rust will be memory safety as long as no other component overwrites its memory). A second way to define properties is vulnerability oriented: by modelling how a vulnerability (e.g. buffer overflow) changes the properties of a component, we can ask MSpec for a configuration that ensures other components' requirements are satisfied - i.e. a configuration which mitigates the vulnerability.

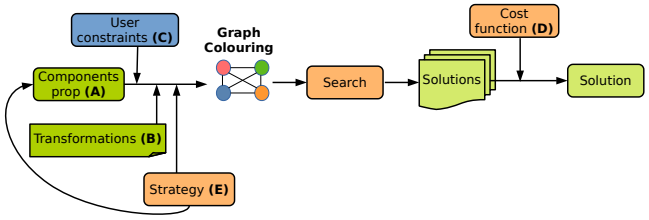

Figure 1. Overview of MSpec

3 Preliminary results

Currently, we have integrated MSpec with FlexOS and are working on safety reasoning and its implications in zero code vulnerability patching. Our tool is able to create configurations where components with different properties (e.g. Rust, Dafny scheduler, plan C) have their safety guarantees hold at runtime. We have measured the research space reduction when using MSpec with FlexOS. In the scenario of four components, MPK and KASAN/Stack Protector/CFI we were able to reduce from thousands of configurations to only two equivalent configurations. A second path we explored is automatic vulnerability mitigation. We were able to mitigate CVE-2014-0160 named Heartbleed by modelling the vulnerability into the DSL of the affected components.

4 Further work

One of the key issue that we plan to tackle is the search space explosion problem. Since compartmentalization mechanisms are partitioning a set of n objects into k nonempty subsets, the complexity is given by Stirling partition number which is exponential. We will explore several algorithms and the usage of a SMT solver for finding safe configurations quickly. Additionally, we will improve MSpec to capture the behavior of mechanisms such as KASAN as well as pre- and post- conditions for function calls. Finally, we plan to integrate MSpec with at least one other popular OS such as Linux to showcase its generality and applicability

5 Related work

Orthogonal to our work, Caramine et al. [2] offers a solid foundation for reasoning about security of practical compartmentalized applications but only targets simple applications written in the programming language defined in the paper. SOAAP [4] proposes a system to explore software's compartmentalization space using static/dynamic analysis; however, this work targets monolithic userspace codebases and has yet to gain real world usage due to its complexity. uSCOPE [10] proposes a least privilege compartmentalization strategy but lacks generality. Enclosure [3] used a DSL based approach to isolate libraries using MPK but only targets interpreted languages such as Python. Zhang et all propose automatic isolation based on CVEs [15]. Tsampas et all [12] propose discuss automatic compartmentalization of C programs on capability machines.

References

[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. 2009. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)* 13, 1 (2009), 1–40.

[2] Carmine Abate, Arthur Azevedo de Amorim, Roberto Blanco, Ana Nora Evans, Guglielmo Fachini, Catalin Hritcu, Théo Laurent, Benjamin C Pierce, Marco Stronati, and Andrew Tolmach. 2018. When good components go bad: Formally secure compilation despite dynamic compromise. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1351–1368.

[3] Adrien Ghosn, Marios Kogias, Mathias Payer, James R. Larus, and Edouard Bugnion. 2021. Enclosure: Language-Based Restriction of Untrusted Libraries. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 255–267. https://doi.org/10.1145/3445814.3446728

[4] Khilan Gudka, Robert N.M. Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G. Neumann, and Alex Richardson. 2015. Clean Application Compartmentalization with SOAAP. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (Denver, Colorado, USA) *(CCS '15)*. Association for Computing Machinery, New York, NY, USA, 1016–1031. https://doi.org/10.1145/2810103.2813611

[5] Volodymyr Kuznetzov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. 2018. Code-pointer integrity. In *The Continuing Arms Race: Code-Reuse Attacks and Defenses*. 81–116.

[6] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Alexander Jung, Stefan Teodorescu, Sebastian Rauch, Felipe Huici, Costin Raiciu, and Pierre Olivier. 2021. FlexOS: Towards Flexible OS Isolation. *arXiv preprint arXiv:2112.06566* (2021).

[7] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Ştefan Teodorescu, Pierre Olivier, Tiberiu Mosnoi, Răzvan Deaconescu, Felipe Huici, and Costin Raiciu. 2021. FlexOS: Making OS Isolation Flexible. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Ann Arbor, Michigan) *(HotOS '21)*. Association for Computing Machinery, New York, NY, USA, 79–87. https://doi.org/10.1145/3458336.3465292

[8] K Rustan M Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 348–370.

[9] Derrick McKee, Yianni Giannaris, Carolina Ortega Perez, Howard Shrobe, Mathias Payer, Hamed Okhravi, and Nathan Burow. [n. d.]. Preventing Kernel Hacks with HAKC. ([n. d.]).

[10] Nick Roessler, Lucas Atayde, Imani Palmer, Derrick McKee, Jai Pandey, Vasileios P. Kemerlis, Mathias Payer, Adam Bates, Jonathan M. Smith, Andre DeHon, and Nathan Dautenhahn. 2021. uSCOPE: A Methodology for Analyzing Least-Privilege Compartmentalization in Large Software Artifacts. In *24th International Symposium on Research in Attacks, Intrusions and Defenses* (San Sebastian, Spain) *(RAID '21)*. Association for Computing Machinery, New York, NY, USA, 296–311. https://doi.org/10.1145/3471621.3471839

[11] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. Addresssanitizer: A fast address sanity checker. In *2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*. 309–318.

[12] Stylianos Tsampas, Akram El-Korashy, Marco Patrignani, Dominique Devriese, Deepak Garg, and Frank Piessens. 2017. Towards automatic compartmentalization of C programs on capability machines. In *Workshop on Foundations of Computer Security 2017*. 1–14.

[13] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. {ERIM}: Secure, efficient in-process isolation with protection keys ({MPK}). In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 1221–1238.

[14] Jonathan Woodruff, Robert NM Watson, David Chisnall, Simon W Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G Neumann, Robert Norton, and Michael Roe. 2014. The CHERI capability model: Revisiting RISC in an age of risk. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, 457–468.

[15] Ze Zhang, Qingzhao Zhang, Brandon Nguyen, Sanjay Sri Vallabh Singapuram, Z Morley Mao, and Scott Mahlke. 2020. Automatic Feature Isolation in Network Protocol Software Implementations. In *Proceedings of the 2020 ACM Workshop on Forming an Ecosystem Around Software Transformation*. 29–34.