Towards a Theory of Application Compartmentalisation

Robert N. M. Watson[1], Steven J. Murdoch[1], Khilan Gudka[1], Jonathan Anderson[1], Peter G. Neumann[2], and Ben Laurie[3]

[1] University of Cambridge
[2] SRI International
[3] Google UK Ltd.

Abstract. Application compartmentalisation decomposes software applications into sandboxed components, each delegated only the rights it requires to operate. Compartmentalisation is seeing increased deployment in vulnerability mitigation, motivated informally by appeal to the principle of least privilege. Drawing a comparison with capability systems, we consider how a distributed system interpretation supports an argument that compartmentalisation improves application security.

1 Introduction

Application compartmentalisation decomposes applications into sandboxed components, each assigned only the rights it requires to operate. Motivated by the *principle of least privilege*, the focus of historic work on compartmentalisation has been primarily in access-control policy enforcement [25]. More recently, compartmentalisation has been employed in *vulnerability mitigation*: exploited vulnerabilities leak only the subset of overall application rights that are held by the compromised sandbox. For example, web browsers might be compartmentalised such that each web page visited is rendered in its own sandbox [22]. Successful exploitation of a JavaScript rendering bug might lead only to very limited leakage of system-centered rights (e.g., local files) and application-centered rights (e.g., username/password tuples for other web sites).

To date, application compartmentalisation has been intuitively grounded in the principle of least privilege, but without a theoretical foundation that permits the use of formal or automated reasoning. In this paper, we consider experience gained in developing and deploying compartmentalised applications, and its implications for new theoretical foundations. Our approach considers applications to be distributed systems, and is, therefore, a fundamentally protocol-centered approach. This viewpoint grants us access to a large existing literature on network and distributed system analysis: we reason about the gains attackers make in communicating with, and compromising, elements of the system as a network of components. One important outcome of this work will be a new approach to application security measurement.

2 Protection Model

Application compartmentalisation is premised on strong *isolation* between individual compartments: no communication is permitted except via controlled communication channels. This is a view long-espoused in security system design, ranging from microkernel- and security-kernel systems to programming-language virtual machines. Isolation deployed within applications typically follows a *sandboxing* model: code is encapsulated in a process or other execution container, and granted only specific rights delegated or forwarded from the containing system. We have previously observed an elegant alignment between the intersection of sandboxing features across operating system platforms and the *capability system model* [7, 26].

Capability system models can be mapped into classic OS primitives (processes and IPC) if access to ambient authority is limited. This approach falls naturally out of classic capability hardware and OS designs such as the CAP computer [27] and seL4 [12], but also in hybrid capability systems such as Capsicum that allow selected processes to operate in a non-ambient "capability mode" [26]. Capability system models can also be layered over other substrates, such as distributed systems or programming languages. Examples of the latter include Joe-E [17] and Caja [18].

All of these systems are able to represent non-hierarchical protection models: mutually distrusting program instances with disjoint sets of rights may safely interact. However, the efficiency of cross-domain calls involving mutual distrust varies significantly – programming languages such as Java provide this very efficiently through a blend of static and dynamic enforcement, whereas hardware-supported process models rely on slower message passing via a mutually trusted kernel. This variation introduces a necessary set of tradeoffs between performance and security – i.e., more granular compartmentalisations that better approximate the principle of least privilege may incur greater cost on some substrates.

In general, we believe that sandboxing schemes approximate capability systems, but with a not-uncommon problem that support for flexible delegation and fine-grained application-level access control may be limited by some substrates (e.g., SELinux with its static rule configuration [16]). It is unclear to us whether this rigidity improves performance; in our experience, however, it observably increases fragility in the presence of ongoing software development.

3 Applications as Distributed Systems

In Capsicum, the kernel and a small amount of userspace communications code act as the run-time Trusted Computing Base (TCB) [2]. Sets of sandboxes and their interconnections are able to represent different communication and trust relationships, including both purely hierarchical relationships (e.g., the HTTPS download component depends fully on the ambient component of `fetch`), non-hierarchical isolation (e.g., different renderer processes in a web browser), and

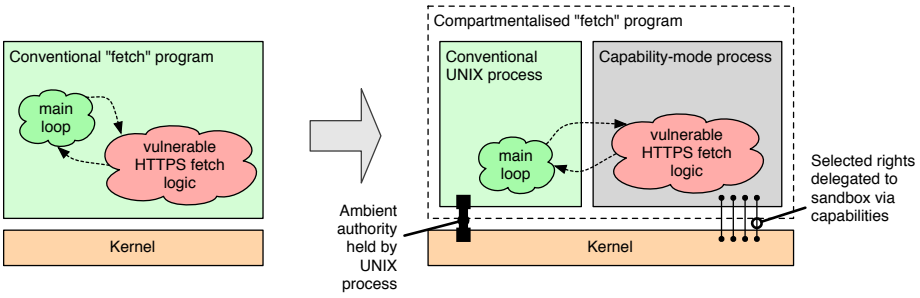

Fig. 1. Whereas conventional HTTPS `fetch` executes within a single process holding ambient user privilege, Capsicum's `fetch` executes TLS in a sandbox holding only delegated rights. This is a code-oriented compartmentalisation: selected risky code runs in a per-application instance sandbox.

non-hierarchical mutual distrust with communication (e.g., two components representing different stages in a firewall processing pipeline). This approach suggests a graph-oriented analysis of program structure, in which nodes execute components with state (processes), and edges are IPC channels (perhaps sockets).

However, this graph captures only communications, and not trust (or perhaps more accurately, dependence), which may track communication edges (especially in a purely information-flow-centric analysis), but also span multiple edges via intermediate nodes. For example, microkernel systems often employ the notion of a service namespace manager, such as in Mach [1]; isolated components will necessarily trust the namespace manager in some form, but via the namespace manager they may indirectly trust the actions of other parties that are reachable via the shared namespace. As such, trust is more complex than simple annotations on communications edges in the graph.

This is fundamentally a distributed system view of application structure, allowing us to borrow an extensive literature on protocols, consensus, fault tolerance, and distrust, including Lamport's Byzantine Generals [13], and more recent work on understanding and managing compromise in distributed systems [23], software composition [19], and layering of compartmentalised software over microkernels and separation kernels [3].

4 Compartmentalisation Philosophies

Figure 1 illustrates the transformation of a conventional application, `fetch`, into a compartmentalised one via Capsicum. The kernel provides a capability system substrate; a portion of `fetch` operates with ambient authority, outside of the capability system, and a sandboxed HTTPS download component executes with only delegated rights. In this example, two types of rights have been delegated

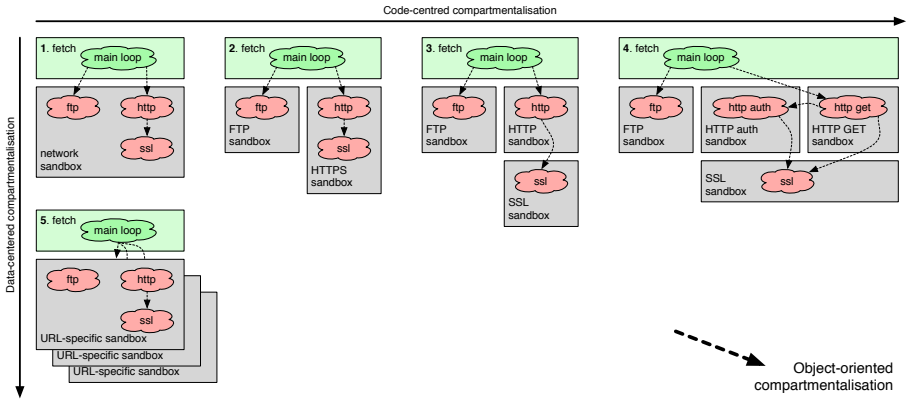

Fig. 2. `fetch` and `libfetch` can be compartmentalised along many different cut points, with different security, performance, and complexity tradeoffs.

via *kernel capabilities*: a set of explicitly delegated files and sockets, and an IPC channel used to communicate with the parent.

Even a program as simple as `fetch` serves as a useful proving ground for exploring ideas about compartmentalisation – not least, by bringing to light the observation that a single program may have many different possible decompositions, with different security properties. The illustrated compartmentalisation is fundamentally *code-oriented*, in that two pieces of code, a main loop and a set of network functions, are separated from one another. Selection of a decomposition is often grounded in our understanding of *past vulnerabilities*: OpenSSL code has suffered a number of past vulnerabilities, both stemming from incorrect implementation and incorrect use. Frequently, these vulnerabilities have been remotely exploitable, leading to remote code execution, which can be mitigated by sandboxing.

However, further decomposition along *data-oriented* lines can also be justified: `fetch` can accept multiple URLs on the command line, and if an exploit originates from one web server communicated with, exploit code may have access to later files downloaded in the same code-oriented sandbox. We might, therefore, choose to further introduce sandboxes one per web site, pursuing the principle of least privilege. We might reasonably take the view that this is an *object-oriented partitioning*, instantiating an object to process each URL, based on a common class, even though the C programming language itself does not capture those higher-level programming properties. It is for this reason that we suggest that fine-grained compartmentalised applications adopt the *object-capability paradigm*; Figure 2 illustrates additional points on the code-based spectrum, including finer-grained compartmentalisations even within the processing of a single HTTPS connection.

We make several further observations about the nature of compartmentalisation. Finer-grained decompositions require tradeoffs between security goals, program complexity, and performance, as security-beneficial decomposition requires both programmer attention and incurs a run-time overhead. In designing decompositions, we are responding to informal notions of risk – properties of the code itself: past vulnerabilities, source code provenance (e.g., open-source supply-chain trojans), and risky code structures (e.g., video CODECs). However, we are also taking into account where data originated (a file or web site), its sensitivity (e.g., keying material), and how it will be processed. Notions of data provenance (informally, taint) and the nature of rights that could be leaked will all be inputs to this reasoning.

These are all aspects of compartmentalisation design that we would like to capture in a structured model.

5 A Graph-Oriented Analysis

Traditional graph representations of networks (whether of connected hosts, collections of applications on a host, or subsystems within an application) represent components as nodes and edges indicate permitted communication paths between components. This representation lends itself to compartmentalisation through blocking communication paths which are not necessary, using firewalls, mandatory access control, or capabilities as appropriate. Eliminating an edge in this model by blocking a communication path will improve security, but not all edges are created equal. Typically, there is some concept of the source of malicious activity (e.g. the Internet), and following connectivity from here to other nodes will show which nodes are at particular risk.

However, merely being connected to a potentially malicious node does not necessarily imply that the network design is vulnerable. While some communication paths are highly dangerous (e.g. exposing an industrial control system designed without security in mind, to the Internet), others may be far less problematic (e.g. connecting a hardened web server to the Internet). Including information about how vulnerable a particular node is can help, as it indicates the likelihood that a node may be compromised if it encounters malicious input. However, even this extension is not sufficient – for example while connecting the web server to the Internet may be fine, connecting its file system to a malicious file server is likely not.

The traditional model can capture connectivity, but not trust, and so has significant limitations when it comes to measuring the network. Instead each node can be modelled as a series of ports, and connectivity is from a port on one node to a port on another, forming a matrix of probabilities. Rather than a single vulnerability probability for a node, there is a probability assigned to each pair of nodes stating the probability that a malicious output will result from a malicious input. In the example of a web server, a malicious input on the socket input of the web server is unlikely to lead to malicious output and thus will be

assigned a low probability, whereas a malicious input in the file system input of the same program will likely result in malicious output on all ports.

This approach captures both trust between nodes and vulnerability of applications, but a high probability does not necessarily mean that an individual node is somehow flawed. A router may forward malicious traffic (unless the router has a suitably configured firewall) even though it is operating as intended. To evaluate whether a network is secure it is necessary to establish the consequences of connectivity between malicious nodes and critical resources to be protected.

Defences can also be modelled, in this approach. For example compartmentalising an application may not affect the probability that it will be compromised, and so will not be captured by assigning a single probability of compromise figure to the node. However, the matrix approach is more suitable – compartmentalisation results in the probability for malicious output will be lowered in cases where the input and output ports are in different compartments.

While powerful, the challenge of using the model is in its complexity. Extracting data to fill in the vulnerability matrix is challenging. Also the computational complexity of reasoning about the network is high, due to the number of network states growing exponentially with the number of states of each node. Therefore new techniques in data collection and simulation will need to be adopted for this model to be fruitful.

6 Related Work

Application compartmentalisation is a recasting of the microkernel hypothesis into the application space – in fact, contemporary monolithic applications are of a similar scale (millions of lines of code) to the monolithic kernels that motivated microkernel research. Past security-kernel research is concerned with providing a reliable TCB for decomposed components [15], and more recent microkernel research has likewise been interested in the verifiability of security properties when combining untrustworthy components over a formally verified separation kernel [3].

Karger originally proposed the use of capability systems to contain trojans [10], an approach later adopted by Provos in SSH privilege separation [21] and Kilpatrick in Privman [11]. While these application decompositions were concerned with UNIX root privilege, contemporary application compartmentalisation is more interested in limiting rights to ambient (unprivileged) user rights, as utilised by Reis et al in Chromium [22] and by the authors in Capsicum [26]. This is a response to the observation that, on a single-user machine, access to the single user's account is, in practice, almost as important as access to root privilege.

We are interested in capturing a variety of trust relationships in application compartmentalisation – not least, hierarchical trust models explored in Multics [24], non-hierarchical models, such as assured pipelines, from Type Enforcement [5], and the flexible programmer-driven models supported by capability sys-

tems that differentiate policy and enforcement, such as in CAP [27], PSOS [20], and Hydra [14].

Research into automation of application decomposition is also directly relevant, although not always well-supported by current theory. Brumley and Song developed Privtrans [6]; Bittau et al, Wedge [4], and most recently, Harris et al have used parity games to drive automata-based application of policies to compartmentalised software [9] – a policy- rather than least-privilege–oriented approach. Our own SOAAP toolchain attempts to take into account many factors in selecting (and trading off) application decompositions in a dialogue with the developer, which has motivated our search for formal grounding [8].

7 Conclusion

Our ongoing work with application compartmentalisation has driven us to begin development of theory helping us to justify and quantify program decompositions. Throughout, the principle of least privilege (together with desires for good software engineering practices such as abstraction, encapsulation, and facile composability) guides our approach, with a focus on providing vulnerability mitigation. In a broad sense, compartmentalisation represents the adoption of further distributed system programming paradigms in local systems: interconnected components are isolated in sandboxes used to construct larger user-facing applications, and subject to a variety of faults (malicious and otherwise). This has led us to a graph-oriented analysis that will provide the foundation for modelling application security through quantifiable comparisons of risk and rights exposure. This in turn will lead to the development of automated tools to help develop and reason about compartmentalisation strategies.

8 Acknowledgments

We would like to thank our colleagues on the Capsicum, CTSRD, and SOAAP projects for their thoughts and comments contribution to this paper, including Ross Anderson, David Chisnall, Brooks Davis, Pawel Dawidek, Steven Hand, Anil Madhavapeddy, Ilias Marinos, Will Morland, Michael Roe, and Hassen Saidi.

We gratefully acknowledge Google, Inc. for its sponsorship. Portions of this work were sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8750-10-C-0237. The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the Department of Defense.

References

1. Accetta, M., Baron, R., Golub, D., Rashid, R., Tevanian, A., and Young, M. Mach: A New Kernel Foundation for UNIX Development. Tech. rep., Computer Science Department, Carnegie Mellon University, August 1986.
2. Anderson, J. P. Computer Security Technology Planning Study. Tech. rep., Electronic Systems Division, Air Force Systems Command, Hanscom Field, Bedford, MA 01730, October 1972.
3. Andronick, J., Greenaway, D., and Elphinstone, K. Towards proving security in the presence of large untrusted components. In *Proceedings of the 5th Workshop on Systems Software Verification* (October 2010).
4. Bittau, A., Marchenko, P., Handley, M., and Karp, B. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation* (2008), USENIX Association, pp. 309–322.
5. Boebert, W. E., and Kain, R. Y. A practical alternative to hierarchical integrity policies. In *In Proceedings of the 8th National Computer Security Conference* (1985).
6. Brumley, D., and Song, D. Privtrans: automatically partitioning programs for privilege separation. In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13* (Berkeley, CA, USA, 2004), SSYM'04, USENIX Association, pp. 5–5.
7. Dennis, J. B., and Van Horn, E. C. Programming semantics for multiprogrammed computations. *Commun. ACM 9*, 3 (1966), 143–155.
8. Gudka, K., Watson, R. N. M., Hand, S., Laurie, B., and Madhavapeddy, A. Exploring compartmentalisation hypotheses with SOAAP. In *Proceedings of the Workshop on Adaptive Host and Network Security (AHANS 2012)* (September 2012), IEEE.
9. Harris, W. R., Farley, B., Jha, S., and Reps, T. Secure Programming as a Parity Game. Tech. Rep. 1694, University of Wisconsin Madison, July 2011.
10. Karger, P. A. Limiting the damage potential of discretionary trojan horses. In *IEEE Symposium on Security and Privacy* (1987), pp. 32–37.
11. Kilpatrick, D. Privman: A Library for Partitioning Applications. In *Proceedings of USENIX Annual Technical Conference* (2003), USENIX Association, pp. 273–284.
12. Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. seL4: formal verification of an operating-system kernel. *Commun. ACM 53* (June 2009), 107–115.
13. Lamport, L., Shostak, R., and Pease, M. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems 4*, 3 (July 1982), 382–401.
14. Levin, R., Cohen, E., Corwin, W., Pollack, F., and Wulf, W. Policy/mechanism separation in Hydra. In *SOSP '75: Proceedings of the fifth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1975), ACM, pp. 132–140.
15. Lipner, S. B., Wulf, W. A., Schell, R. R., Popek, G. J., Neumann, P. G., Weissman, C., and Linden, T. A. Security kernels. In *AFIPS '74: Proceedings of the May 6-10, 1974, National Computer Conference and Exposition* (New York, NY, USA, 1974), ACM, pp. 973–980.

16. Loscocco, P. A., and Smalley, S. D. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proceedings of the USENIX Annual Technical Conference* (June 2001), USENIX Association, pp. 29–42.
17. Mettler, A., Wagner, D., and Close, T. Joe-E: A Security-Oriented Subset of Java. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2010* (February 2010).
18. Miller, M. S., Samuel, M., Laurie, B., Awad, I., and Stay, M. Caja: Safe active content in sanitized javascript, May 2008. `http://google-caja.googlecode.com/files/caja-spec-2008-06-07.pdf`.
19. Neumann, P. G. Principled assuredly trustworthy composable architectures. Tech. rep., Computer Science Laboratory, SRI International, Menlo Park, December 2004.
20. Neumann, P. G., Boyer, R. S., Feiertag, R. J., Levitt, K. N., and Robinson, L. A Provably Secure Operating System: The System, Its Applications, and Proofs, Second Edition. Tech. Rep. CSL-116, Computer Science Laboratory, SRI International, May 1980.
21. Provos, N., Friedl, M., and Honeyman, P. Preventing privilege escalation. In *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12* (Berkeley, CA, USA, 2003), SSYM'03, USENIX Association, pp. 16–16.
22. Reis, C., and Gribble, S. D. Isolating web programs in modern browser architectures. In *EuroSys '09: Proceedings of the 4th ACM European Conference on Computer Systems* (New York, NY, USA, 2009), ACM, pp. 219–232.
23. Robertson, P., and Laddaga, R. Adaptive security and trust. In *Proceeings of the Workshop on Adative Host and Network Security* (Septmeber 2012), IEEE.
24. Saltzer, J. H. Protection and control of information sharing in Multics. In *SOSP '73: Proceedings of the fourth ACM Symposium on Operating System Principles* (New York, NY, USA, 1973), ACM.
25. Saltzer, J. H., and Schroeder, M. D. The protection of information in computer systems. *Proceedings of the IEEE 63*, 9 (September 1975), 1278–1308.
26. Watson, R. N. M., Anderson, J., Laurie, B., and Kennaway, K. Capsicum: Practical capabilities for UNIX. In *Proceedings of the 19th USENIX Security Symposium* (Berkeley, CA, USA, 2010), USENIX Association.
27. Wilkes, M., and Needham, R. *The Cambridge CAP Computer and Its Operating System*. Elsevier North Holland, New York, 1979.