Exploring compartmentalisation hypotheses with SOAAP

Khilan Gudka*, Robert N. M. Watson*, Steven Hand*, Ben Laurie† and Anil Madhavapeddy*
**University of Cambridge - first.last@cl.cam.ac.uk*
†*Google UK Ltd. - benl@google.com*

***Abstract*—Application compartmentalisation decomposes software into sandboxed components in order to mitigate security vulnerabilities, and has proven effective in limiting the impact of compromise. However, experience has shown that adapting existing C-language software is difficult, often leading to problems with correctness, performance, complexity, and most critically, security. Security-Oriented Analysis of Application Programs (SOAAP) is an in-progress research project into new semi-automated techniques to support compartmentalisation. SOAAP employs a variety of static and dynamic approaches, driven by source code annotations termed *compartmentalisation hypotheses*, to help programmers evaluate strategies for compartmentalising existing software.**

***Keywords*-Privilege separation, sandbox, compartmentalisation, program analysis, capability system, object capabilities.**

I. Introduction

This paper introduces the Security-Oriented Analysis of Application Programs (SOAAP), a set of tools to support semi-automated compartmentalisation of C-language Trusted Computing Base (TCB) components such as operating systems and web browsers. Application compartmentalisation is the decomposition of software into multiple sandboxed components, each granted only the rights it requires to operate, in order to mitigate security vulnerabilities. Compartmentalisation has proven to be a powerful technique, and is used in applications ranging from OpenSSH [1] to the Google Chrome web browser [2] to limit the rights available to an attacker after a successful exploit.

However, compartmentalising software has proven quite difficult. A previously "local" application becomes a distributed one, linked by a web of IPC channels, with the programmability and debuggability challenges that implies. Compartmentalisation necessarily changes the behaviour of a program, but ideally only in cases where unexpected code (e.g., injected via a buffer overflow attack) runs, rather than in typical use. In our own prior work on Capsicum [3], we experienced this first-hand, discovering that even simple decompositions led to subtle bugs.

SOAAP is a set of analysis and transformation techniques to help programmers introduce compartmentalisation into existing C-language applications. There are many possible decompositions that may (or may not) accomplish a variety of security goals, which must be traded off with the correctness, performance, and complexity penalties of compartmentalisation. Programmers use C-language annotations, termed *compartmentalisation hypotheses*, to evaluate possible decompositions without actually implementing them. SOAAP reports potential bugs and interactions, helping the programmer iteratively refine their compartmentalisation plan. Such tools are critical to the further deployment of compartmentalisation, whether it is with more conventional operating system sandboxing systems such as Capsicum, or experimental systems such as SRI International and the University of Cambridge's CHERI CPU architecture [4].

This paper introduces SOAAP early in the project life cycle, and is intended to seek feedback from the security research community. We propose a variety of approaches, discuss our current implementation, and consider potential evaluation techniques to use as the work matures.

II. Application compartmentalisation

The end-user software ecosystem consists of an operating system kernel, hundreds of libraries, a window server, language runtime environments, and web browsers, which themselves include language interpreters, virtual machines, and rendering engines. Collectively, this TCB consists of many tens of millions of lines of trusted (but not trustworthy) C and C++ code. Coarse hardware, OS, and language security models mean that much of this code is security-critical: a single flaw, such as an errant NULL pointer dereference in the kernel, can expose all rights held by the user to an attacker. The consequences of compromise are serious, and include loss of data, release of personal or confidential information, damage to system and data integrity, and even total subversion of a users online presence by the attacker.

Vulnerabilities are widespread – the National Vulnerability Database reported 16 new serious vulnerabilities a day during the week of 16 July 2012 [5] – but there can be delays of weeks or months before they are patched, and further delays before users upgrade, leaving systems at risk. Further, patches themselves may introduce new vulnerabilities.

Compartmentalising applications – breaking them into components that run in sandboxes granted only selected rights – has proven effective in mitigating security vulnerabilities. Abstractly, the approach appeals to the *principle of least privilege*: following a compromise, only authority held by the successfully exploited component is available to the attacker, rather than the full rights of the application.

Figure 1 illustrates a compartmentalisation of the `gzip` command-line compression tool using Capsicum: vulnerable

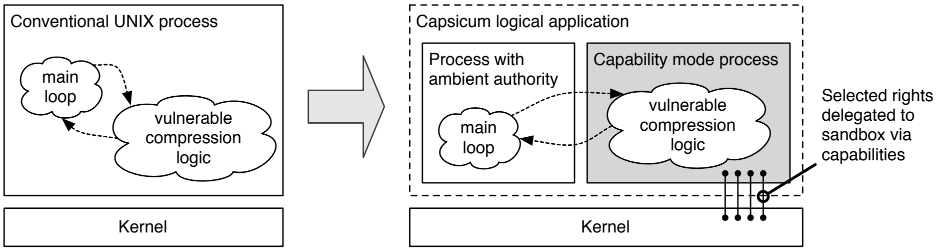
Conventional UNIX process
main loop
vulnerable compression logic
Kernel
Capsicum logical application
Process with ambient authority
main loop
Capability mode process
vulnerable compression logic
Selected rights delegated to sandbox via capabilities
Kernel

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

compression code is placed in a sandbox and delegated only required rights to source and target files, rather than the more traditionally held *ambient authority* to all objects accessible to the user. This technique is applied in Google's Chrome web browser, placing HTML rendering and JavaScript interpretation into sandboxes isolated from the global file system. Compartmentalisation is the only technique that we are aware of that can successfully mitigate both *known* and *unknown* classes of vulnerabilities, since it is not specific to attack techniques, and is therefore one we would like to see more widely deployed.

III. The compartmentalisation problem

Despite its clear and widely described benefits, compartmentalisation proves highly problematic in systems combining the UNIX process model and C/C++ runtime environments. Applications must be converted to employ OS message passing rather than using a unified address space for communication between components, sacrificing programmability and performance by making a local programming problem into a distributed systems one.

As a result, large-scale compartmentalised applications are difficult to design, write, debug, maintain, and extend, raising serious questions about correctness, performance, and most critically, security. A simple example arose in Capsicum: we introduced a bug when compartmentalising `gzip` because we did not propagate the compression level command-line argument value to the sandbox: the global variable was assigned to after the sandbox was `fork()`ed and thus did not see the update, leading to incorrectness.

In addition to bugs in program correctness, bugs can also prevent compartmentalisation from accomplishing its goals: we have previously argued [6] that it is not the size of the TCB that indicates how secure it is but the number and ways untrusted code can interact with it. If sandboxes are allowed to perform arbitrary privileged operations by sending messages to trusted processes, then compartmentalisation has not provided any gains in security.

Validating that a compartmentalised program continues to operate as originally intended, while also confirming that compartmentalisation introduced the intended security benefits, justifying security-performance tradeoffs, is a key challenge to further deployment of compartmentalisation.

IV. SOAAP

In SOAAP, we are exploring and addressing these critical problems, cleaning up the process and mechanisms of compartmentalisation. At a high-level, the goal of SOAAP is to allow application programmers to more easily and strategically trade-off performance, complexity, and security through semi-automated analysis of possible compartmentalisations. This is done through the iterative annotation and refinement of program source code using SOAAP's static and dynamic analysis tools, as illustrated in Figure 2.

A. Functional correctness

The compartmentalised version must preserve intended program behaviour, while reducing the set of allowed executions corresponding to malicious activity – a process made tricky because the only resource available is the (presumed vulnerable) application source code. Today, this validation is done through manual source code review and casual testing. Such a process is error prone as even a missed data-dependency can lead to significant functional bugs.

B. Hypothesis exploration

SOAAP presupposes the existence of a large set of possible application decompositions structured around the flow of data and code over time, which select not just different tradeoffs between performance, complexity, and security, but also

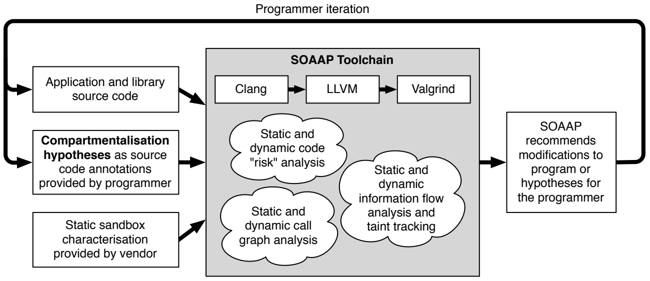
Programmer iteration
Application and library source code
Compartmentalisation hypotheses as source code annotations provided by programmer
Static sandbox characterisation provided by vendor
SOAAP Toolchain
Clang
LLVM
Valgrind
Static and dynamic code "risk" analysis
Static and dynamic information flow analysis and taint tracking
Static and dynamic call graph analysis
SOAAP recommends modifications to program or hypotheses for the programmer

Figure 2. SOAAP's engages the software developer in an iterative cycle of compartmentalisation hypothesis development and testing.

different levels of effort during compartmentalisation. This level of investment is a key factor in compartmentalisation strategy: once a library or application has been sandboxed in one way, it can be difficult and time-consuming to re-compartmentalise in order to accomplish new security goals.

At one end of the spectrum, all application components run in a single address space with no separation enforced by the execution substrate. At the other end, the principle of least privilege is enforced throughout: every function (or even block of code) executes only with access to the memory and services required to perform its function.

SOAAP allows different *compartmentalisation hypotheses* to be explored without paying the full price of performing compartmentalisation; it can also be used to find bugs in already-compartmentalised applications. Programmer annotation of hypotheses drives static and dynamic analysis, reporting both potential bugs in correctness (e.g., non-introduction of data consistency bugs), and whether or not the annotated set of security goals is met (e.g., information flow constraints). Annotations can describe which methods should run in a sandbox, which global state and file descriptors are allowed to be read or written by a sandbox, how the platform's sandbox facilities limit access to system services, and what rights are available via RPC interfaces. Given these annotations, SOAAP can identify missing data dependencies, list unaccounted for system calls, enumerate rights delegated to sandboxes, and warn about security bugs such as information improperly leaked into sandboxes.

C. Automatic inference

Key to compartmentalising an application is understanding the risks of its activities: compartmentalisation overhead should be invested only where there is useful benefit. However, understanding risk in code is tricky – it involves expert intuitions about the nature of the application, and significant knowledge of vulnerability classes and their history. For example, protocol parsing, image processing, and data compression have all proven rich sources of exploitable vulnerabilities. Therefore, an important goal of SOAAP is to incorporate not just static notions of source code risk (e.g., pointer arithmetic) but also dynamic information flow analyses to determine which risky code is exposed to untrustworthy sources. For example, this might help avoid the complexity overhead of sandboxing poorly written but unexposed configuration file parsing, while still highlighting sandboxing opportunities for packet parsing.

D. Iterative compartmentalisation

We believe that there is a useful comparison to be made between compartmentalising software and introducing parallelism into software through multi-threading: both involve analyses of information flow, points of interaction, and the programmability and debuggability risks of distributed computation. In parallel programming, APIs such as pthreads [7] and OpenMP [8] allow programs to be incrementally parallelised as programmers iteratively identify performance bottlenecks through profiling, optimising thread count and synchronisation. The entire program does not need to be made multi-threaded at once in order to gain significant performance improvement – moreover, parallelising the entire application at once is error-prone and almost intractable.

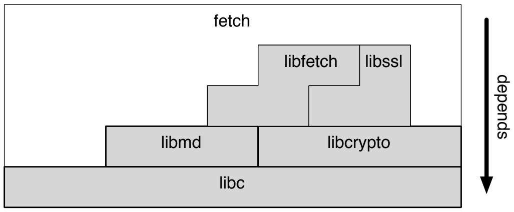

Figure 3. The FreeBSD `fetch` utility is a thin wrapper around cryptography libraries and `libfetch`, a library supporting FTP, HTTP, HTTPS, and local file retrieval. `libfetch` in turn relies on OpenSSL in order to implement TLS. The bulk of code implementing many applications is in large part in supporting libraries.

With SOAAP, we hope to afford similar incremental development benefits: programmers iteratively annotate security goals and compartmentalisation opportunities, addressing additional complexity only as they attempt to accomplish additional security goals. SOAAP analysis will substitute for profiling by providing feedback on potential incorrectness both with respect to original program functionality and compartmentalisation goals. This approach appears promising as programs do not need to be entirely decomposed at once: a single mitigated vulnerability in a sandbox gives an immediate improvement, so compartmentalisation effort can be focused on the most critical areas.

V. FETCH AND LIBFETCH

To illustrate our ideas in SOAAP more concretely, we will use FreeBSD's `fetch` utility and associated `libfetch`; its dependencies are illustrated in Figure 3. `fetch` is a command-line tool for downloading files over FTP, HTTP, and HTTP with TLS, and is used for package download, and supports HTTP authentication and proxy servers. `libfetch` encapsulates all protocol-specific communication; this is a common structure – the majority of code in many applications comes from component or vendor-provided class libraries. `fetch` is an interesting example for several reasons:

- It is a general-purpose tool run with full user privilege.
- Processing URLs and HTTP headers is a source of past vulnerabilities with significant exposure (i.e., to untrusted web servers).
- OpenSSL, used for cryptographic processing, has had significant past vulnerabilities.
- File servers and proxy servers may require authentication data that is, itself, sensitive.

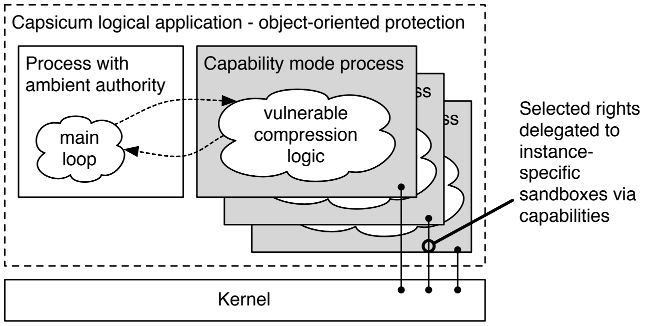
Capsicum logical application - object-oriented protection
Process with ambient authority
main loop
Capability mode process
ss
ss
vulnerable compression logic
Selected rights delegated to instance-specific sandboxes via capabilities
Kernel

Figure 4. In object-oriented compartmentalisation, compressed files are processed in separate sandboxes. If a vulnerability is exploited by a zipped file from one origin, it no longer has access to data from another origin.

- The interests of many different parties meet: the originator of file content, the server sourcing the file, any intermediate proxy servers, and the local user.
- Fetch may process multiple URLs at a time, each interacting with different end-files, servers, etc.

With such a diverse array of potentially exploitable vulnerabilities and mutually untrusting parties, `fetch` encapsulates in a simple program (and supporting libraries) much of the design-space tradeoff of monolithic applications such as mail readers, web browsers, and office suites.

VI. Exploring the compartmentalisation space

`fetch` and `libfetch` are sufficiently complex (roughly 6KLOC) to support several different compartmentalisation strategies, addressing different aspect of `fetch`'s behaviour, represented interests, and attacker models. The compartmentalisation philosophy calls for a minimisation of privilege assigned to each compartment, which requires identifying elements of the application with security sensitivity – access to system objects, sensitive data, etc.

Past privilege separation work (e.g., OpenSSH) and policy-related access control work (e.g., SELinux) has focused to a greater extent on the exposure of system-maintained objects such as sockets and files. However, we are interested in finer-grained aspects of program operation not visible to the operating system, such as the exposure of authentication or keying material, and enforcing processing pipeline independence, rather than treating untrustworthy portions of applications as a single black box.

Likewise, we are interested not just in sandboxing HTTP processing in `fetch`, but protecting integrity and confidentiality between separate instances of downloads – preventing the leaking of data of a file downloaded from one site back to another site just because both URLs are fetched by a single execution of `fetch`. This suggests an object-centered compartmentalisation rather than a purely code-centered one; Figure 4 illustrates how the compartmentalisation of `gzip` using a single sandbox might be revised.

Figure 5 illustrates a spectrum of code-centered and data-centered sandboxing strategies for `fetch` and `libfetch`. On the X-axis, increasing sandboxing granularity for code components offers successful attacks decreasing ability to influence processing in other components. On the Y-axis, introducing new compartments for successive URLs prevents independent file retrievals from influencing or leaking the outcomes of other requests. The illustrated cases are:

1) A single network service sandbox for `libfetch` limits access to delegated files.
2) By separating FTP and HTTPS processing from one another, vulnerabilities in one code path are prevented from influencing processing in the other.
3) Further separation of HTTP from SSL processing prevents leakage of a client authentication key in the SSL sandbox into a compromised HTTP sandbox.
4) Decomposition of HTTP into two phases, authentication and GET, prevents a vulnerability in file download from allowing access to a proxy password.
5) Isolating download sessions prevents information about earlier downloads from being leaked into a compromised sandbox, or corrupting later downloads.

As granularity increases across both dimensions, the result is an object-oriented compartmentalisation: isolation is along functional boundaries that capture self-contained program elements – classes – and between different object instances, preventing undesired interactions along either dimension. This style of decomposition is well-aligned with the object-capability security model promoted by compartmentalisation systems such as Capsicum, which support ephemeral sandboxing and flexible delegation at run-time, but less-well suited to statically configured systems such as SELinux [9] – a restatement of an observation made in Capsicum, in which the shipped SELinux policy for Chrome placed different renderer instances were conflated in a single renderer domain, rather than independent and ephemeral sandboxes.

VII. Incremental sandboxing and testing

Given code- and data-oriented compartmentalisation goals, SOAAP allows the programmer to specify annotations capturing certain security objectives:

- Functions that should run sandboxed.
- Global state that should be accessible in sandboxes.
- Descriptors that can be read/written to by sandboxes.
- System calls accessible to sandboxes.
- Privileges available via RPC interfaces.

In essence, SOAAP annotations capture a sandboxing policy within source code, keeping it close to the implementation and in the vocabulary of the application programmer. A static/dynamic tool can then be used to validate whether the proposed sandboxing would violate policies captured by annotations (e.g. reading or writing a disallowed global variable; or performing a disallowed operation on a file descriptor, such as `chmod()`).

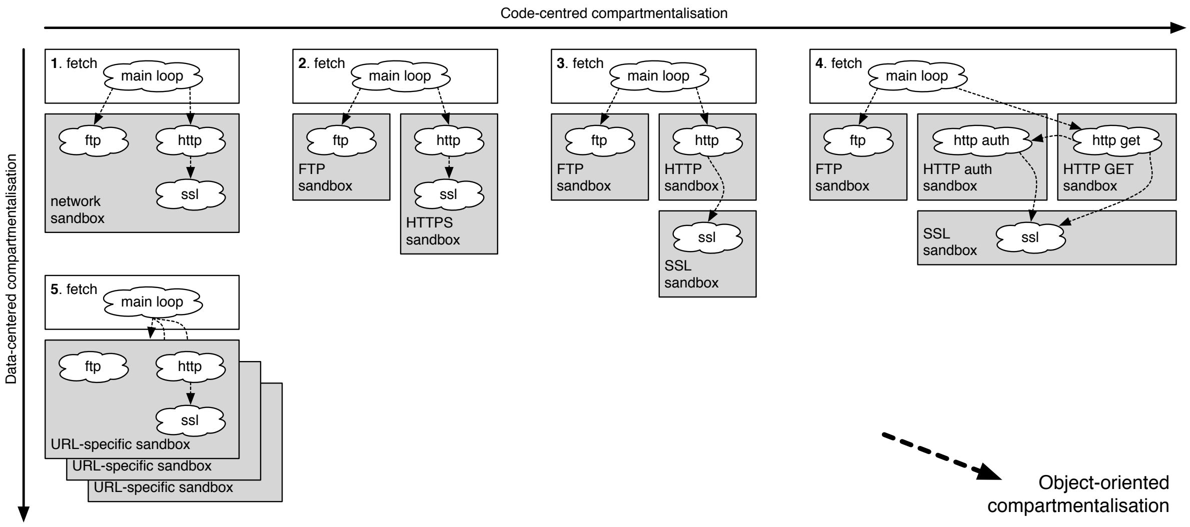

Figure 5. `fetch` and `libfetch` can be compartmentalised along many different cut points, with different security, performance, and complexity tradeoffs.

A necessary input to these tools is a characterisation of the sandboxing technology, be it Capsicum, Seccomp, etc – this will inform enforcement and performance tradeoff analysis. This approach supports a narrative in which a programmer incrementally identifies their security goals, annotating methods to run in a sandbox, then identifying state that should (or should not) be shared through iterative executions of SOAAP tools. Once a desirable and consistent compartmentalisation is identified, it can then be implemented using techniques such as Capsicum.

VIII. The tool

We have modified Clang/LLVM to implement new C-level annotations that declare compartmentalisation hypotheses. These annotations are checked statically using LLVM and dynamically using Valgrind to validate functional correctness, identify any behaviours that have not been specified (e.g. reading from an unauthorised global variable, or performing an undeclared file descriptor operation), validating whether sandboxes are truly isolated (i.e. can information flow unintentionally between sandboxes), and inferring what privileges sandboxes hold through direct delegation or via messages to/from other processes. Thus SOAAP tests whether the proposed compartmentalisation would preserve the intended behaviour of the original program and also analyses the security implications, providing useful debugging output to the programmer.

A. Annotations

SOAAP implements the following annotations:

- **__sandbox_persistent**, **__sandbox_ephemeral** on a C function indicating that it should run in a sandbox. The former indicates that the sandbox should be created during process linkage (i.e., before `main()`), and persist across invocations. The latter creates a fresh sandbox each time the function is called.
- **__readvar**, **__writevar** on global variables; indicates that the variable can be read and/or written from sandboxes.
- **__readfd**, **__writefd** on file descriptor function arguments; indicates that a delegated file descriptor can be read or written.

LLVM compiles these C annotations to Valgrind client requests, which allow information to be communicated to Valgrind at execution-time. A programmer then runs the compiled program with the SOAAP-extended Valgrind, which performs dynamic checking as the program executes. Wherever possible, our goal is to check properties statically, but due to the unsoundness of C, effects of inter-process compartmentalisation, and the implications of library use, we check some properties dynamically. For example, we dynamically check for leakage of data between ephemeral sandboxes. In the next two sections, we illustrate how these annotations are used and how SOAAP is able to find some bugs pertaining to both functional correctness and isolation.

B. Checking Functional Correctness

Figure 6 shows a simple contrived C program that performs `gzip`-like compression. The example has been greatly abstracted to keep the focus on annotations and finding bugs using them.

The `main()` method opens the input and output files and passes them to `compress()`, which performs the necessary compression depending on the required compression level indicated by `cflag`. Compression algorithms are a well-known source of vulnerabilities and thus the programmer hypothesises that this should run in a sandbox. For performance reasons, they decide to have a sandbox that persists across all requests, thus the **__sandbox_persistent** annotation is used. They also annotate that the sandbox is only allowed to read from the `in` file and write to the `out` file using the **__readfd** and **__writefd** annotations respectively. As `cflag` is needed, they also annotate that the sandbox should be allowed to read from it using **__readvar**. The programmer tries these set of annotations as a first attempt and compiles and runs it using SOAAP. SOAAP's output is shown in Figure 7.

We identify two bugs. Firstly, that the write to the global variable `cflag` will not be propagated to the sandbox, as a **__sandbox_persistent** sandbox will have been created before `cflag` is set from command line arguments – if the sandbox mechanism is `fork()`-based, this bug might easily go unnoticed (and was during early Capsicum experiments with `gzip`!) as the linkage is still satisfied and compilation would have succeeded. The second bug identifies that the global variable `vflag` is read from the sandbox but has not been annotated as such. This could either be an access that is not supposed to be allowed, in which case the programmer may decide to move the access outside of the sandbox. Otherwise, it could be that they forgot to annotate it – in a `fork()`-based programming environment, linkage would have been satisfied but a stale value would have been used if no explicit forwarding over IPC had been implemented.

C. Validating Isolation

SOAAP can also identify when persistent sandboxes may leak heap memory between subsequent entries into the sandbox. Consider the example sample program in Figure 8. This is a stripped down version of the HTTP authentication code performed by `fetch`, abstracted to highlight the annotations and bugs we can find.

A persistent sandbox is created at the start of the main process and is shared across all invocations of the functions that are to run within them. Thus, if heap memory is allocated and even freed, if it is not overwritten then subsequent invocations executing in the same sandbox could potentially access it. If such memory contained sensitive information that only that invocation is allowed to see, then there could be a breach of isolation and in such cases, ephemeral sandboxes would give stronger guarantees. When running this program with SOAAP, we produce the output shown in Figure 9.

```
1  int __readvar cflag = 0; // compression level
2  int vflag = 0; // verbose flag
3
4  int main(int argc, char** argv) {
5      int in = open(argv[1], O_RDONLY);
6      int out = open(argv[2], O_WRONLY | O_CREAT);
7      cflag = ...;
8      vflag = ...;
9      compress(in, out);
10     close(in);
11     close(out);
12     return 0;
13 }
14
15 __sandbox_persistent
16 int compress(int __readfd in, int __writefd out) {
17     // 1. read from in ...
18     // 2. compress depending on cflag value
19     switch(cflag) {
20         ...
21     }
22     // 3. write to out ...
23     if (vflag) {
24         // write some stats to stdout
25     }
26     return 0;
27 }
```

Figure 6. A simple `gzip`-like compression program that takes an input and output file argument and optional flags for the compression level and verbosity. This listing is abstracted to highlight the annotations and how they can help a programmer find bugs in their compartmentalisation hypothesis.

```
*** Warning ***
Global variable "cflag" is
written to in method main (compress.c:7)
after a sandbox has been forked, so the
updated value will not be seen.

 at 0x8048827: main (compress.c:7)

*** Warning ***
Sandbox read global variable "vflag" in
method compress (compress.c:23), but it
is not allowed to.

 at 0x8048892: compress (compress.c:23)
 by 0x8048416: _start1 (in bin/compress)
 by 0x8048387: (below main) (in bin/compress)
```

Figure 7. The results of running the simple annotated compress program through SOAAP.

IX. PROPOSED EVALUATION DIRECTIONS

SOAAP is at a very preliminary stage but already with simple analyses, we are able to find some useful bugs (including one that we actually encountered during earlier work on compartmentalisation) thus showing that there is significant potential benefit from tools of this kind. Furthermore, the programmer can change the sandboxing policy simply by changing the annotations making the testing cycle very quick. As SOAAP progresses, we will incorporate more sophisticated analyses of information flow to track the movement of sensitive data as well as characterisations of sandboxing mechanisms to inform the programmer of exactly what guarantees they can expect.

As the work matures, we anticipate evaluating it from a number of perspectives:

```
int main(int argc, char** argv) {
    ...
    int socket = ...;
    char* user = ...;
    char* pass = ...;
    http_basic_auth(socket, user, pass);
    ...
    return 0;
}

__sandbox_persistent
void http_basic_auth(int __writefd sock, char* u, char* p) {
    char* auth;
    asprintf(auth, ”Authorization Basic: %s:%s”, u, p);
    write(sock, auth, strlen(auth));
    free(auth);
}
```

Figure 8. A `fetch`-like program for performing HTTP basic authentication. SOAAP identifies that there is a memory leak between subsequent invocations of `http_basic_auth()`.

```
*** Warning ***
Persistent sandbox allocates heap memory,
which is accessible by subsequent requests.
Consider using an ephemeral sandbox.

 at 0x5AF75: malloc (vg_replace_malloc.c:266)
 by 0xF04DA: vasprintf (in /lib/libc.so.7)
 by 0xEF57D: asprintf (in /lib/libc.so.7)
 by 0x80488EC: http_basic_auth (http_auth.c:14)
 by 0x80488A7: main (http_auth.c:6)
```

Figure 9. The results of running the HTTP authentication program through SOAAP.

- How do false positive and negative rates arising out of the unsoundness of C-language program analysis affect the user experience?
- When applied to a back catalogue of known compartmentalisation bugs, are all found, and if not, why not?
- Are new bugs found in previously compartmentalised programs, illustrating the benefits of this approach?
- Once a viable and desirable compartmentalisation is identified, and then implemented by the programmer, are there other problems that arise?
- Do performance predictions made by SOAAP prove accurate?
- Can we scale up SOAAP-based exploration to both very large collections of programs, such as the footprint of a complete UNIX system, or individually large (monolithic) applications such as web browsers and mail clients?
- Although grounded in the vocabulary and problem space of C-language software, similar types of problems have arisen in the compartmentalisation of software in other languages (most notably Java) – can our techniques be transposed to that space?
- What is the resulting compile-time and run-time overhead of SOAAP analyses? How well do they scale and are there optimisations to reduce their space-time requirements? Although this overhead is only encountered during analysis, large running time and memory requirements will impede their use and the speed of compartmentalisation hypothesis iterations.

X. Related work

The *principle of least privilege*, which demands that computations run only with the privileges they require, along with the core security goals of protecting confidentiality, integrity, and availability, is first enumerated by Saltzer and Schroeder's 1975 article *The Protection of Information in Computer Systems*. Karger's 1987 article on trojan horse mitigation [10] lays the conceptual groundwork for privilege separation, and later application compartmentalisation, which became mainstream techniques applied to system-level applications in the early 2000s with Provos's work on OpenSSH [1], and Kilpatrick's Privman [11]. Application compartmentalisation is later applied to user-level applications by Reis et al. in the Chrome web browser [2] and by Watson et al. in Capsicum [3], where the focus is on intra-application security concerns rather than system privileges. All of these projects have reported on the difficulty in applying compartmentalisation to C-language applications; however, in this work we draw particularly on our own experiences with Capsicum.

Automatic privilege separation has been discussed in the literature – Brumley and Song's Privtrans [12] and Bittau's Wedge [13] both explore techniques for assisting the programmer in identifying and exploiting compartmentalisation opportunities. Privtrans takes a code-oriented view, focusing on dividing operation between privileged and unprivileged processes through program annotation, whereas Wedge relies on programmer-provided memory type information. Harris's secure programming by parity games [14] reasons about the defense characteristics of Capsicum-based application compartmentalisation where policies are represented by automata checked on the interactions of compartments.

SOAAP builds significantly on these approaches by providing semi-automated tools promoting an object-capability philosophy explored in HYDRA [15], and in many ways comparable to Mettler et al.'s imposition of object-capability semantics on Java via Joe-E [16] – albeit in the unconstrained execution environment of UNIX processes rather than a type-safe and already object-oriented language runtime or virtual machine, requiring the imposition of an object-oriented structure. SOAAP additionally incorporates analysis techniques grounded in information flow, past vulnerability information, and source code risk analysis, allowing object-capability boundaries to align with critical security constructs in the application and its library footprint. Rather than implementing separation policies, SOAAP is an exploration tool for programmers who may not fully understand the code that they are separating, recognising the complexity of large-scale applications with hundreds or thousands of contributors and large third-party library footprints.

XI. Conclusion & future work

This paper has described SOAAP, a set of techniques and tools for exploring *compartmentalisation hypotheses* in support of application compartmentalisation for vulnerability mitigation. Despite promising early results, SOAAP remains an in-progress research project. We anticipate a number of future directions, including enriching the annotation scheme to allow labelling of data *sensitivity* and code *riskiness*, automatic inferring of sandboxing opportunities, applying SOAAP to analysing existing hand-compartmentalisations for bugs, and automated techniques for implementing hypothesised compartmentalisations. We also hope to apply SOAAP to finer-grained compartmentalisations on the CHERI processor, allowing us to add additional dimensionality to our static sandbox characterisations and performance tradeoff analyses. Finally, a significant further effort will occur in evaluating our approaches, including the degree to which security benefit arises from real-world use, and the accuracy of hypothesisation testing and results.

Acknowledgment

Thanks are due to Peter G. Neumann, Wei Ming Khoo, Jonathan Anderson, and Pawel Jakub Dawidek for their helpful contributions and feedback on this paper.

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] N. Provos, M. Friedl, and P. Honeyman, “Preventing privilege escalation,” in *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*, ser. SSYM’03. Berkeley, CA, USA: USENIX Association, 2003, pp. 16–16.

[2] C. Reis and S. D. Gribble, “Isolating web programs in modern browser architectures,” in *EuroSys ’09: Proceedings of the 4th ACM European Conference on Computer Systems*. New York, NY, USA: ACM, 2009, pp. 219–232.

[3] R. N. M. Watson, J. Anderson, B. Laurie, and K. Kennaway, “Capsicum: Practical capabilities for UNIX,” in *Proceedings of the 19th USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2010.

[4] R. N. Watson, P. G. Neumann, J. Woodruff, J. Anderson, R. Anderson, N. Dave, B. Laurie, S. W. Moore, S. J. Murdoch, P. Paeps, M. Roe, and H. Saidi, “CHERI: a research platform deconflating hardware virtualization and protection,” in *Workshop paper, Runtime Environments, Systems, Layering and Virtualized Environments (RESoLVE 2012)*, March 2012.

[5] NIST Computer Security Resource Center, “National vulnerability database,” 2012, http://nvd.nist.gov/.

[6] D. G. Murray and S. Hand, “Privilege separation made easy: trusting small libraries not big processes,” in *Proceedings of the 1st European Workshop on System Security*, ser. EUROSEC ’08. New York, NY, USA: ACM, 2008, pp. 40–46.

[7] B. Nichols, D. Buttlar, and J. P. Farrell, *Pthreads programming*. Sebastopol, CA, USA: O’Reilly & Associates, Inc., 1996.

[8] OpenMP Architecture Review Board , “OpenMP Application Program Interface Version 3.1,” Tech. Rep., 2011.

[9] P. A. Loscocco and S. D. Smalley, “Integrating Flexible Support for Security Policies into the Linux Operating System,” in *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, June 2001, pp. 29–42.

[10] P. A. Karger, “Limiting the damage potential of discretionary trojan horses,” in *IEEE Symposium on Security and Privacy*, 1987, pp. 32–37.

[11] D. Kilpatrick, “Privman: A Library for Partitioning Applications,” in *Proceedings of USENIX Annual Technical Conference*. USENIX Association, 2003, pp. 273–284.

[12] D. Brumley and D. Song, “Privtrans: automatically partitioning programs for privilege separation,” in *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, ser. SSYM’04. Berkeley, CA, USA: USENIX Association, 2004, pp. 5–5.

[13] A. Bittau, P. Marchenko, M. Handley, and B. Karp, “Wedge: Splitting Applications into Reduced-Privilege Compartments,” in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association, 2008, pp. 309–322.

[14] W. R. Harris, B. Farley, S. Jha, and T. Reps, “Secure Programming as a Parity Game,” University of Wisconsin Madison, Tech. Rep. 1694, July 2011.

[15] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf, “Policy/mechanism separation in Hydra,” in *SOSP ’75: Proceedings of the fifth ACM Symposium on Operating Systems Principles*. New York, NY, USA: ACM, 1975, pp. 132–140.

[16] A. Mettler, D. Wagner, and T. Close, “Joe-E: A Security-Oriented Subset of Java,” in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2010*, February 2010.