Building the Loom Framework on FreeBSD

BY BRIAN KIDNEY

When trying to understand code in production, developers log information to be analyzed offline. This will usually require the decisions up front as to what and when to log. The answers to these questions are usually based on a determination of where things could go wrong or previous issues experience in the system. Knowing during development exactly what information is needed during development is not always possible, especially when trying to trace data for security reasons.

This is part of the problem faced by the Causal Adaptive Distributed, and Efficient Tracing System (CADETS) research project[1]. The goal of the project was to use existing mechanisms as well as develop new techniques to instrument FreeBSD to maximize transparency into live servers. This would give users better insight into security concerns on their systems, as well as provide additional information for performance tuning and debugging.

Loom was one of the tools developed as part of this effort. In this article, we will look at the original inspiration for building Loom. We will talk about how Loom has been expanded beyond its original purpose. Finally we will discuss what we are working on for the future of Loom.

In this article, we will look at the original inspiration for building Loom.

Instrumenting for Security

When developers instrument their software, it is usually to track performance or trace possible points of failure in a system. Similar methodology is often used when adding security monitoring, often capturing only a small number of events such as login attempts with minimal information. Much of this information is logged to different locations on a computer and often is not correlated with other systems. The CADETS team found that this information in the kernel could be obtained easily with the use of DTrace, the tracing framework included as part of the FreeBSD base system. FreeBSD provides many DTrace providers to allow access to information on kernel components such as system calls, function calls and network buffers.

However, extracting additional information from userland processes DTrace requires more work. The issues with tracing programs running in userland is the information captured often lacks context and flexibility. For example, it is easy to write a DTrace script to log all calls to a specific function in a library, but the script has to be applied each process that will use that library. We needed a way to be able to instrument the library itself so that the logging occurs for any program using the library without having to target each program explicitly in the script. The mechanism needed also to provide provide context for each time the library function was called, including function name, arguments and executable calling the function.

DTrace provides a mechanism to add tracing probes directly into programs and librar-ies via Userland Statically Defined Tracing (USDT). The process for creating these probes requires the developer to write and compile custom code for each probe, include it in the original source and then use a DTrace specific tool to modify the programs binary object files to insert calls to the probes. Unfortunately, this process requires changes to the build process to include an additional tool that modifies your object files directly. Each time the probe code is changed it is converted to a header file that is included in the original code, requiring the program to be completely compiled from scratch. We wanted a simpler sys-tem that allows the user to insert instrumentation at the LLVM Intermediate Representation (IR) stage, without the need for a complete rebuild. Since FreeBSD uses LLVM as the system compiler this would make it easier to include our method in the build system.

Our solution to this problem was Loom, a custom LLVM optimization (opt) pass that al-lowed the user to insert arbitrary code into a program during compile time without modifi-cation of the original source code and without having to recompile the entire program each time. As you can see in Figure 1, Loom integrates into the FreeBSD build process as an ad-ditional opt pass. The original source code is compiled to LLVM IR first and using the Loom pass and a YAML policy file the additional code is inserted into the program before the final linker stage.

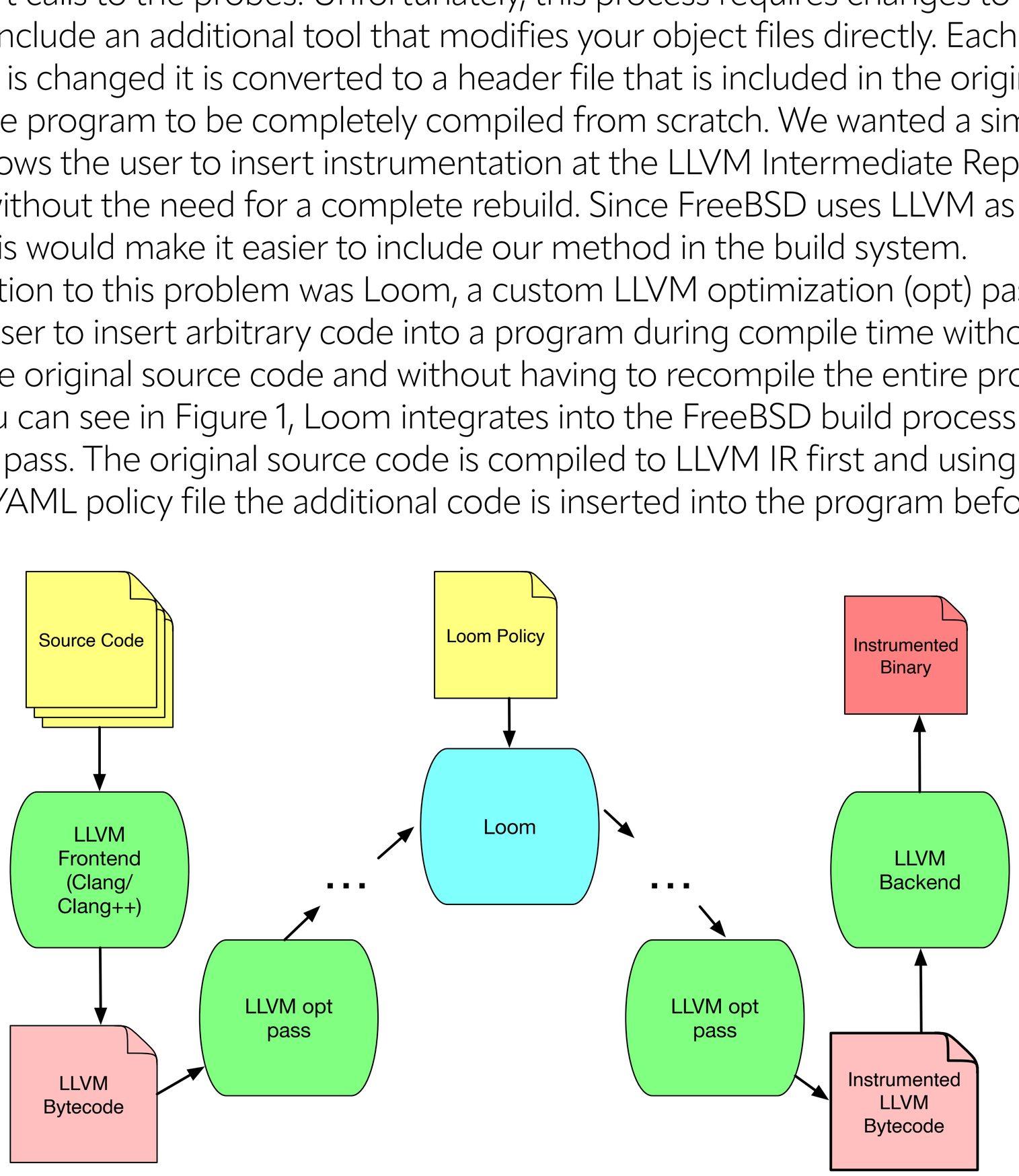

Figure 1: Loom Build Process

Unlike the original USDT method, changes to source code are not needed. In fact, the source code of the application is not needed. As long as a LLVM IR representation of the code is available, the instrumentation can be added to the program.

One CADTES use case is to capture all authentication attempts using the Pluggable Authentication Modules (PAM) system. These can come from many sources such as SSH or sudo, so to capture them all we targeted the PAM library itself. A sample Loom policy YAML file is given in Listing 1 showing how we where able to instrument PAM to log authentication attempts from userspace. The example is specifically instrumenting the pam_authenticate function, capturing all of the arguments to the call. Metadata is used to add context to the data when it is received by DTrace, allowing the author of the corresponding DTrace script to differentiate data from various userland probes. As a result, we are able to capture the username (and other details) of login attempts and the executables from which they come.

```
strategy: callout
dtrace: userspace
functions:
- callee: [entry]
  metadata:
    name: auth
    id: 1
  name: pam_authenticate
```

Listing 1: Loom configuration for logging PAM authentication attempts

In the majority of cases, this solution does not require the user to write custom code as in USDT. To get data into the DTrace system, an experimental system call was written using the SDT provider from DTrace to pass trace data to the kernel. The policy file provides all of the information needed to add context to the user data when tracing. The only time additional development is needed is when the user wants to transform the data in some way before it is sent to DTrace.

Using Loom allowed us to reproduce the DTrace USDT functionality without having to integrate the USDT toolchain into FreeBSD build process. The system does require a custom system call but it was a simple change that provided the flexibility to add and remove instrumentation to applications without any modifications to the source code. Since the operating system already uses LLVM we could integrate Loom into the build by adding additional build targets for programs and libraries to produce the LLVM IR output. Then Loom could be called as an additional build step where needed.

In the majority of cases, this solution does not require the user to write custom code as in USDT.

The FreeBSD base system contributed to the ease of this development. The base system of the operating system not only includes the kernel and userland source to be able to run a fully functional operating system, but it also includes a unified system for building FreeBSD. In order to use Loom in our research project we needed to be able to build program and libraries as bit code objects (BCOs). These BCOs are a binary form of LLVM IR that Loom can modify for instrumentation and transformation. Since all binaries in FreeBSD use a central set of build scripts we only had to modify these to create BCOs for any part of the base system. Once we had made these modifications we could apply Loom to any program or library necessary.

Expanding Loom

For some CADETS use cases it was necessary to transform the data collected before passing it to DTrace. For example, to avoid name collision between servers in an instrumented distributed system the project uses Globally Unique Identifier (GUID) for the usernames so they could be individually tracked in the outputs. To achieve this, a mechanism was added to allow external code to be inserted into a program through Loom. The user can add calls to custom functions or libraries as long as the symbols are available at the link stage of the build process. Though this functionality was originally designed to modify data before logging it, the concept opened up new possibilities such as code transformation in addition to instrumentation.

Since the conclusion of the CADETS project, we have been working on exploring these possibilities, such as the replacement of code within a program to use a new API. For example, to test a new network API would traditionally require changes to source code, replacing calls to the old API with calls to the new one. This process is tedious and prone to human error. We are expanding Loom to handle such tasks without the need for source code changes. By matching a set of function calls, we can have Loom remove the original code and replace it with one or more calls using of the new API.

Loom is a full featured instrumentation and transformation framework.

One current limitations to this work is the policy file which is used to specify the changes that Loom needs to make. Though we have the ability to make code transformations with Loom, the current YAML based format is not expressive enough to fully specify transformations. We are currently working on a language that will overcome this limitation. The aim of this language is to allow for very specific specification of transformations to be made by Loom.

The Future of Loom on FreeBSD

Loom is a full featured instrumentation and transformation framework. Loom has the ability to instrument functions and function calls as well as accesses to structure fields, global variable and pointers. One to one function call replacement is fully implemented and there is functionally to replace calls to a sequence of functions, though further configuration work is required to make this generally usable. Additionally there is the ability for many of these configuration to be matched using wildcards or limited within the scope of certain files from the source code.

Since its use in the CADETS project, Loom has seen interest from developers on other operating systems such as Linux. Though we have made efforts to support these users, Loom's main development continues on FreeBSD. With the upcoming addition of Link Time Optimization (LTO) in the FreeBSD build system, we will investigate the possibility of using Loom in the unmodified FreeBSD build system. We will also be use FreeBSD to test the new Loom configuration language, investigating areas where the transformation system can help maintain ports of software from other operating systems.

For more information on Loom and to follow future developments you can check out the project page at github.com/cadets/loom.

Acknowledgements

The author would like to thank those who helped and guided me during this work including George Neville-Neil and Domagoj Stolfa for their DTrace help. Thanks to Ed Maste for his answers to my FreeBSD questions whether it was directly or to connect me with someone who knew the answer. Many thanks to all the members of the CADETS Project including Robert Watson, Arun Thomas, Silviu Chiricescu, Jon Anderson and Amanda Strand.

The FreeBSD community was great help to our efforts. Whether it was on IRC, mailing lists or at BSD conferences, it was generally easy to connect with the community and get answers to questions when we ran into issues. Thanks to the members of the community who were more than happy help out.

Finally the author would like to thank Jon Anderson for his feedback to improve the manuscript.

This work has been sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8650-15-C-7558. The views, opinions, and/or findings contained in this paper are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Department of Defense or the U.S. Government.

[1] Anderson, J; Neville-Neil, G. V.; Thomas, A.; Watson, R. N. M. "Cadets: Blending Tracing and Security on FreeBSD," *FreeBSD Journal*, May/June 2017, 12 - 17. 5

BRIAN KIDNEY is an Instructor of Cybersecurity at the College of the North Atlantic in St. John's, Newfoundland, Canada. He is also completing a PhD in Computer Engineering at Memorial University. His research interests include privacy and security, specifically as they relate to operating systems and programming languages. Brian has 20 years of experience as a Software Engineer developing software for multiple industries.