Peter Lotts

Adding network subsystem provenance collection to CADETS

Computer Science Tripos – Part II

Downing College

April 24, 2018

Proforma

Name:	**Peter Lotts**
College:	**Downing College**
Project Title:	**Adding network subsystem provenance collection to CADETS**
Examination:	**Computer Science Tripos – Part II, 2018**
Word Count:	**tbc**[1]
Project Originator:	Dr R. Sohan
Supervisor:	Dr G. Jenkinson

Original Aims of the Project

To provide additional metadata collection tools for network packets to the Computer Laboratory's existing CADETS project. This will be provided by tracking individual packets as they flow through the network stack of the FreeBSD[2] kernel, and making information regarding memory locations used and the time taken by each layer to process a given packet available to DTrace[3]. DTrace is a generic Dynamic Tracing framework which is the primary data collection tool used by the CADETS project to bring kernel data into userspace for analysis. The performance impact of the project on network packet delivery is to be evaluated.

Work Completed

What we actually did.

Special Difficulties

None. [We hope!!]

[1] This word count was computed at [date] by the provided online tool at `URL`

[2] `http://www.freebsd.org/`

[3] `http://dtrace.org/`

Declaration Of Originality

I, Peter Lotts of Downing College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed [signature]

Date [date]

Contents

List of Figures

Chapter 1

Introduction

1.1 Use Case

In this project I am evaluating the application of fine-grained tracing within the network stack of the FreeBSD kernel by implementing additional statically defined DTrace probes which are able to provide a platform for both security and performance analysis.

The security aspect of the tool is most useful in combating Advanced Persistent Threats (APTs)[4], where the attacker slowly infiltrates the target system, hiding out of sight until they can exploit knowledge gained to access the enterprises internal network. This may allow the attacker to extract sensitive data from databases in such a way that the access is very hard to distinguish from normal access. Eventually, it is likely that the malware will make a mistake and trigger an indicator of compromise to be observed by administrators, but by then it is difficult to tell what data the malware has seen.

The Computer Laboratory has a project which is trying to combat this by building CADETS[1] on top of the FreeBSD operating system, which tracks the provenance of data by collecting metadata from computers all over the network about what processes act on what data and when. This metadata is then collected in a distributed database, where it can be analysed to trace data flows throughout the computer system.

My project adds support for collecting metadata on network packets as they flow through the kernel network stack. The data collected will allow users of CADETS to seek out suspicious activity which may be being used to attack a system, and will be able to provide a list of locations in kernel memory where packet data were stored. The latter allows a system administrator to infer what data may have been leaked if they are able to determine that some malicious code had access to a particular set of kernel memory addresses.

The performance analysis platform is provided by DTrace's provision of high-accuracy timestamps on probe firing, which when they are collated for a given packet allow the time spent in each section of the network stack to be evaluated. This could be assessed by system administrators, and may allow for better fine-tuning of protocol parameters, some of which are notoriously difficult to set.

1.2 The Network Stack

Almost all modern general purpose computer networks (including, significantly, the Internet) are built upon a layering of several protocols on top of each-other with well-defined interfaces connecting them, defined by the OSI model[2]. These interfaces are often quite general and so this model allows for different protocols to be used for each layer as desired, somewhat interchangeably; this also provides code separation between modules (i.e. layers) with different duties, and so make implementations of these layers - the so-called 'Network Stack' - easier to maintain.

1.2.1 Layer 1: Physical

The bottom-most layer of the network stack, this layer defines the physical communications medium and how to transmit bit streams over it. Properties such as timing, leading to latency and bandwidth, are mostly defined in this layer, although higher layers are likely to decrease bandwidth somewhat by adding mandatory per-packet header data. Generally these days this layer is 802.11 ('Wi-Fi') or Ethernet, although Ethernet networks generally do not need to handle shared medium communications any more.

1.2.2 Layer 2: Data Link

This layer defines how data frames are transferred between two physically connected nodes, including any shared medium access arbitration and means of addressing such physically connected nodes.

1.2.3 Layer 3: Network

Typically implemented by the Internet Protocol (IP), addressing and routing between physical networks is defined here, typically including concepts such as broadcast (all nodes receive message) and multicast (a specific group of nodes receive message). This layer often has to handle problems arising from the underlying layers having a different Maximum Transmission Unit (MTU), such that a large packet may not be able to make the next hp connection as a single packet if it is too large for the next physical network. In IPv4 this is handled using fragmentation, where the large packet is split up into several smaller ones which are then sent separately, and the receiver must keep a copy of fragments it receives until it can re-assemble the whole packet and deliver it to the layer above. IPv6 addresses the problem by dropping the packet and sending a notification back to the sender.

1.2.4 Layer 4: Transport

This layer is responsible for the reliable delivery of data (if required) across a layer 3 link, and creates the notion of a connection which is opened, used to transmit/receive data, and then closed. This is the most common place for application designers to make a choice about layer implementation - TCP or UDP. TCP (the Transmission Control Protocol) provides reliable, in-order delivery of data to higher layers, along with trying to

provide fairness between connections through adaptive transmission rate control to avoid congestion. Its main alternative, UDP (the User Datagram Protocol), does not provide reliable delivery but in removing this feature is often able to operate with lower latency than TCP.

1.2.5 UNIX Sockets API

Under the UNIX 'everything is a file' abstraction, layer 4 connections are provided to user applications through sockets, objects which are opened and closed in a similar manner to files and which yield a file descriptor for I/O operations while the connection is open. This abstraction is provided by the kernel (meaning all layers described previously are implemented either in hardware or the kernel) and used by other kernel components as well as all userspace applications which perform network communication.

1.3 Performance Issues Tracing the Network Stack

When tracing any application, it is important to consider the performance impact of the tracing system on the target application. Timings can be critical to correct operation of the application, but even when this is not the case it would still be all too easy to impact system performance to the point where it no longer meets specifications. Causing the system to degrade to this point undermines the purpose of the tracing in the first place - namely to run all the time the application is running in the real world to gather data which is later used for security and performance analysis.

Of all applications, it is of the utmost importance not to slow down the operation of the Operating System kernel beyond what is necessary. Every application running on the computer must interact with the kernel to perform I/O to files, networks and so forth, and so slowing down the kernel will impact on all applications which use it. This is especially true of regions which operate under mutual exclusion and so the usual modern-day approach of shifting to multicore may not be able to help.

With regards to timings affecting system operation, parts of the network stack are particularly vulnerable to latency, as timers are integral to their operation. One example of such a protocol is TCP, where Padhye[3] has shown that the throughput of a TCP connection is inversely proportional to the latency which TCP observes on the network connection. Clearly, any undue addition to this latency by low-level network code could have dire consequences for application throughput on the system.

1.4 Similar Work

Just need to mention Resourceful? - I haven't used it so far, is it even relevant?

1.5 Overall design

The project is to assign uniquely identifying tags to each packet as it flows through the network stack, noting that packet fragmentation/reassembly will make this more difficult. This will then allow DTrace tracepoints to read the tag on each packet when an interesting operation (broadly speaking, a memory allocation) is performed on it. From here, scripts written in DTrace's D language will be able to forward information to a DTrace consumer to display results and answer user queries.

Chapter 2

Preparation

2.1 Starting Point

The CADETS project currently has a user-space application which collects metadata on kernel-level datastructures via libdtrace, translates the metadata to a JSON format for easy interpretation, and then sends this away for processing (often over the network). It also has an application with a user interface to display the data it has collected.

The FreeBSD kernel provides a means of tagging its main internal structure of interest, namely `struct mbuf` using `struct mbuf_tags`. It is thought that this will be sufficient to tag a packets data with an unique identifier in order to track its progress through the kernels network stack. `struct mbuf` is used by the FreeBSD kernel as a generic fixed size memory buffer (hence the name) to store network packet data as it passes from the sockets API all the way to physical network interfaces. The structures have the ability to be chained together in a linked-list like structure to store variable amounts of data, and packets are edited in place where headers must be added and so forth.

Under Linux, the Resourceful framework is able to collect data from auto-generated tracepoints within the Linux kernel with relatively low overhead, but it does not currently inspect the network subsystem.

2.2 Installing the Development System

I had no prior experience with FreeBSD usage or development, so the first challenge was to get a system up and running to be used for both development and testing. I decided to host FreeBSD as a virtual machine on my Windows 10 laptop, as it has reasonable performance for compiling code, whilst being portable and allowing me to work on the project wherever I am. The virtual machine was set up using Oracle™ VM VirtualBox[1], as I have most experience using this virtualisation application.

The FreeBSD website provides preinstalled images[2] for various virtual machine types, and I decided to use one of these as a basis due to my inexperience with the operating system. Only slow progress was made during the first few days of using this new operating

[1] `http://www.virtualbox.org/`

[2] `http://www.freebsd.org/where.html#download`

system, as the preinstalled images have a rather basic toolset available and I had to learn how to use a new package manager, and the FreeBSD 'ports' system. Having acclimatised to these, I was able to fork the CADETS custom kernel, build and install it onto the running machine.

2.3 FreeBSD kernel study

The first significant piece of planned project work was to inspect the FreeBSD kernel source code for the network stack, gain some familiarity with it, and note the locations of operations which are of particular interest to the project, either from a packet tagging perspective or from the perspective of packet data being copied to a new memory location.

The study commenced at the point where outgoing packets enter the IP layer of the network stack, namely the top of the `ip_output` function. From here, the packet was followed down through the IP layer, exploring all possible control paths [3] and possible exit routes into lower layers. Once the packets had left to go to device drivers, the study then turned to the `netisr` system, which is responsible for receiving packets from device drivers in the FreeBSD kernel and handing them on to the appropriate next layer (usually the IP layer) to be processed. From here, packets were followed back up the network stack in a similar manner, up to the end of `ip_input`, where packets leave the IP layer to go to the next layer up.

At this point, it was noticed that the project was getting a little behind schedule and, in a meeting with my supervisor, it was decided that for the moment the project would not look any higher than the IP layer, as there was no significant academic benefit to be gained from continuing up to look at the TCP layer[4]. This could be completed at a later stage in the project if time became available.

2.4 Challenges of kernel development

It is generally accepted that kernel development is more difficult than application level development, and that productivity is lower as a result. This is in part due to the language choices involved - the FreeBSD kernel is written in C and there is no way to change that whereas in application code it is possible to use the most appropriate language for the job at hand; at least as high level as C++, anyway. Kernel APIs tend to have more opaque documentation as the focus is on documenting the exact specifications of a call, rather than getting people to use it quickly. Internet searches are often less productive as fewer people have ever encountered your exact situation before, and of course extra care must be taken to understand every line of code written, as any memory access error will result

[3] With one or two notable exceptions, namely that

- Berkley Packet Filter (BPF) was considered to be out of scope for the project
- IPSec allows arbitrary code to be executed via its 'hooks' system, and so this cannot be inspected

[4] In order to associate packets with sockets, however, a small amount of tracing would have to be added to the TCP layer.

in the kernel crashing - and it is much harder to debug the kernel than it is to debug a userspace application.

Although I had experience in C/C++ prior to starting this project, I had no experience with the FreeBSD operating system from either a user's perspective or a developer's. I had occasionally compiled the Linux kernel, but had only done any work on it far enough to fix compile errors. Given this situation, reading through the kernel source code was a non-trivial exercise, as I was not familiar with any of the kernel data structures in use and so had to research these when I encountered them. The pain of doing this is greatly reduced by the existence of Robert Watson's FreeBSD Kernel Cross-Reference[5], which allows one to search for identifiers throughout the kernel source tree. It does not compare to some modern IDEs for development, but as these were not readily available on my development machine the online system was sufficient.

[5] http://fxr.watson.org/

Chapter 3

Implementation

Chapter 4

Evaluation

Chapter 5

Conclusion

Bibliography

[1] Causal, adaptive, distributed, and efficient tracing system (cadets).

[2] *ISO/IEC 7498-1*.

[3] Padhye. Modeling tcp throughput: A simple model and its empirical validation.

[4] Tankard. Advanced persistent threats and how to monitor and deter them.

Computer Science Tripos

Part II Project Proposal Coversheet

Please fill in Part 1 of this form and attach to the front of your Project Proposal.

Part 1

Name:	Peter Matthew Lotts	CRSID:	pml43
College:	DOW	Overseers:	JGD/DJG

Title of Project: Adding network subsystem provenance collection to CADETS

Date of submission: _____ **Human Participants will be used?** No

Project Originator: Dr R. Sohan **Signature:** _____

Project Supervisor: Dr G. Jenkinson **Signature:** _____

Director of Studies: Dr R. Harle **Signature:** _____

Special Resource Sponsor: _____ **Signature:** _____

Special Resource Sponsor: _____ **Signature:** _____

Signatures to be obtained by the Student.

Part 2

Overseer Signature 1: _____

Print Form

Overseer Signature 2: _____

Overseers signatures to be obtained by Student Administration.

Overseers Notes:

Part 3

SA Date Received: _____ ***SA Signature Approved:*** _____

Introduction and Description of Work

Nowadays, most cyberattacks from nation state actors do not take on the traditional form of a piece of malware which tries to find a store of sensitive data and dump it back to the attacker as quickly as possible before system administrators have time to do anything. Instead, these attacks slowly infiltrate the target system, hiding out of sight and learning the use patterns of normal users. It then exploits this knowledge to slowly make horizontal transfers within the target enterprise's internal network, and extract sensitive data from databases in such a way that the access is hard to distinguish from normal access. Such an attack is called an Advanced Persistent Threat (APT), and can go on for months or years without system administrators realising. Eventually, it is likely that the malware will make a mistake and trigger an indicator of compromise to be observed by administrators, but by then it is difficult to tell when and how the malware entered the system, and what data it has seen. This can make impact assessments, as well as efforts to prevent a similar attack in future, very difficult.

The Computer Laboratory has a project which is trying to combat this by building CADETS, a Causal, Adaptive, Distributed, and Efficient Tracing System. The system is being built to track the provenance of data by collecting metadata from computers all over the network about what processes act on what data and when. This metadata is then collected in a distributed database, where it can be analysed to trace data flows throughout the computer system. This should allow analysts to trace malware back to possible entry times and methods, which they can then analyse further. This project will add support for collecting metadata on network packets as they flow through the kernel network stack.

Starting Point

The CADETS project currently has a user-space application which collects metadata on kernel-level datastructures via libdtrace, translates the metadata to a json format for easy interpretation, and then sends this away for processing (often over the network). It also has an application with a user interface to display the data it has collected.

The FreeBSD kernel provides a means of tagging its main internal structure of interest, namely *struct mbuf* using *mbuf_tags*. It is thought that this will be sufficient to tag a packet's data with an unique identifier in order to track its progress through the kernel's network stack.

Under Linux, the *Resourceful* framework is able to collect data from auto-generated tracepoints within the Linux kernel with relatively low overhead, but it does not currently inspect the network subsystem.

Substance and Structure of the Project

The objective of this project is to make additions to the DTrace tracing system under FreeBSD to allow it to trace the flow of data through the network subsystem of the FreeBSD kernel, and allow this data to be integrated with the wider CADETS project. The project should be able to track kernel memory allocations which occur as a result of network packet flow, along with times taken for the different network layers to be traversed and attempting to estimate the cause of delays, such as scheduling conflicts. This extension will allow DTrace probes to access the new packet metadata, and it will therefore be accessible to the existing CADETS userspace application which gathers data for processing.

CADETS already generates a locally unique identifier for each socket opened by a process under observation. In order to track packets through the network stack, this project will need to tag each packet with a further unique identifier, and maintain an association between this identifier and that of the related socket. The FreeBSD kernel already provides a method for tagging kernel structures called *mbuf*s, which hold the data payload of a packet as it flows through the network layers. These can be used as the mechanism for attaching an identifier to each packet. The significant complexity in this area of the project might come from the generation of locally unique identifiers to form tags, as this process must be carried out in a timely fashion in order to prevent unacceptable delays in the processing of network packets. The FreeBSD kernel provides methods for generating unique identifiers, but their performance will have to be assessed to determine whether they are fast enough to be of use. If they are not, then an alternative system, based on existing open source algorithms, will need to be developed.

Metadata collection will be performed by extending DTrace's existing collection schemes, aiming to modify this code rather than writing native kernel code where possible, as this will increase portability across new kernel versions. Once this data is made available to DTrace, the CADETS userspace code should be able to collect it for further processing. This userspace code may need to be modified a little, as it may be most useful for events pertaining to each packet to be reconciled together and presented as a summary record, whereas the incoming data from DTrace will be split according to which CPU core the kernel executed the tracepoint on.

To evaluate the performance impact that the project's trace points are having on the performance of the kernel's network subsystem, the network interface throughput will be measured at the socket level without any monitoring, then with DTrace enabled and collecting general network information but without this project's additions, and then with DTrace collecting data from this project's additional code. This will allow the performance drop to be estimated in context with the performance impact of running DTrace.

Extensions

If the core part of the project is completed in sufficient time, there are several extensions which could be implemented:

1) A simple visualisation tool could be produced as a DTrace consumer in place of the current CADETS userspace application. This would allow the results of monitoring to be viewed in a user-friendly format.
2) Based on trace output from using the full tracing system, the most commonly executed pieces of new code could be found and optimised, if they are deemed to be adversely affecting performance. It may be necessary to use some of the concepts from the *Resourceful* project if DTrace itself is deemed to be causing significant overhead. Any *Resourceful* code will need to be ported from Linux to FreeBSD if it is to be used directly.
3) The existing CADETS user interface could be extended to allow the newly collected packet data to be viewed, and to allow users to dig into the flow of data from a particular socket which is under investigation.

Success Criterion

The project will be successful if it is able to collect metadata on the data flows within the FreeBSD kernel's network stack, associating the journey each packet makes with the socket to which it is linked. This should be achieved without incurring an unacceptable performance overhead for the delivery of network packets.

Plan of Work

The main bulk of the project (after the proposal has been formally accepted) will be split into 14 2-week work packages, with a final week of contingency time at the end of the project before the dissertation submission deadline. Note that the work package over the Christmas period is extended to 3 weeks to allow for inevitable time away from the project during this time. I am planning to take lecture courses with a fairly even split between the Michaelmas and Lent terms, so the workload should be quite even throughout the project. For project management purposes, the expected milestones from each work package will be entered into Bitbucket's issue tracker, allowing progress to be estimated at every stage of the project.

Before 20/10/2017

Find project supervisor and discuss ideas, submit Phase 1 Report form to Overseers and gain their initial approval. Make contact with the Computer Laboratory group developing the CADETS project and meet to discuss ideas. Further discussion with Overseers regarding details of the project. Submit draft proposal, and continue to refine this based on comments from Overseers. Submit final proposal by 12:00 20/10/2017.

21/10/2017 – 03/11/2017

Following acceptance of the project proposal, the research phase of the project can commence. This will involve studying how DTrace is used, and how it can be extended to add more trace points yielding more information. The expected output from this process is some small DTrace scripts, written in DTrace's D language, to collect data about sockets by the methods currently provided by DTrace.

04/11/2017 – 17/11/2017

With some familiarity of DTrace and what is possible, the implementation of the FreeBSD kernel can now be inspected to identify where the kernel should have tracepoints added, what information these would make available ad whether this is sufficient, and where each packet's *mbuf* can have its identifier generated and be tagged. The expected output of this process is a list of references to kernel source lines which look to be good candidates for adding instrumentation.

18/11/2017 – 01/12/2017

The remaining research is into the UUID generation algorithms available in the kernel. Studying these may require manual benchmarking code to be written to assess their performance, and therefore whether they would be suitable for tagging packets. If they are determined to be too slow, then other algorithms will need to be researched and one selected and implemented. Once an algorithm has been chosen, this can be used to implement packet tagging in the kernel.

02/12/2017 – 15/12/2017

Leave Cambridge. Work on adding tracepoints at the locations previously identified, making use of the tags which packets now have attached to them in the kernel. At the top of the kernel stack, make sure that packet identifiers can be associated with their socket.

16/12/2017 – 05/01/2018

Set up useful DTrace probes using new tracepoints to collect data for import into CADETS. This is the phase where any required modifications in the userspace CADETS code will be made to rationalise the incoming data from the different CPU core streams.

06/01/2018 – 19/01/2018

Return to Cambridge. Begin writing progress report and run throughput benchmarking in the three different configurations in order to make a performance assessment of the project code.

20/01/2018 – 02/02/2018

Use results from benchmarking to inform decisions about any project extensions which may be completed, and whether further code optimisation is needed. Complete progress report and send to Supervisor and Overseers. Progress report submission deadline: 12:00 02/02/2018

03/02/2018 – 16/02/2018

Complete functionality of core project. Perform any major refactoring which is deemed necessary, and begin to look at the possible extensions to the project.

17/02/2018 – 02/03/2018

Contingency time for core project / time to work on project extensions. Begin collecting ideas about dissertation including code samples to be used.

03/03/2018 – 16/03/2018

Contingency time for core project / time to work on project extensions.

17/03/2018 – 30/03/2018

Leave Cambridge. Code should by now be functional and approaching its final form. Work on dissertation commences in earnest. Write Introduction section, and begin writing framework for the Preparation and Implementation sections. Code completion deadline: 30/03/2018.

31/03/2018 – 13/04/2018

Complete Preparation and Implementation sections of dissertation from existing frameworks. Finish collecting data for the Evaluation section if need be. Code should be complete, barring small aesthetic changes found to be necessary when writing the Implementation section of the dissertation.

14/04/2018 – 27/04/2018

Return to Cambridge. Complete Evaluation and Conclusions sections of dissertation, meeting with supervisor in person to discuss the current draft of the whole dissertation. Start to check over the required sections around the dissertation itself (table of contents, bibliography, etc).

28/04/2018 – 11/04/2018

Make final changes to dissertation with project supervisor, and ask Director of Studies to read and comment on it. From this, the final copy of the dissertation may be produced.

12/04/2018 – 18/04/2018

Contingency time in case the dissertation is not yet fully complete. This time may be filled with final alterations or more significant writing if this is required. Deadline for dissertation submission: 12:00 18/04/2018.

Resources Declaration

The languages used for this project are entirely determined by the existing code with which the project will be integrating. The kernel module component will have to be written in C, to match the FreeBSD kernel, and DTrace scripts must be written in the D scripting language. This project does not depend on specific hardware, but will require an installation of FreeBSD to test on. I will use a virtual machine for this as I do not have easy access to a computer natively running FreeBSD, and it is easy to take regular backups of a virtual machine's hard drive.

For ease of access, I intend to use my own computer for most development and as the host machine for my virtual testing. It's specifications are: Intel i5-5200U Processor (2.2GHz dual core with hyperthreading), 16GB RAM, Windows 10, VirtualBox 5.1.28 at time of writing (updates will be applied throughout the project as they become available). I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure. Once the virtual machine is set up, a full backup of the host machine will be taken to external media, allowing its state to be restored to the project start state.

All code will be subject to version control using *git*, and the repository will be pushed to the Bitbucket cloud service regularly. Periodically, a copy will also be taken onto the MCS file space, ensuring that a copy is always available in Cambridge. Any files not committed to git will be backed up regularly to external media.