*.CANAL

Static source code analysis tool for early detection of privilege escalation vulnerabilities in C files

Adin Horovitz, Andrew Pellitieri, Eugene Kolodenker,
Josh Joseph, Max Li, Onur Sahin

Boston University, Boston, MA
{ahorovit, kierke, eugenek, joshmd, maxli, sahin}@bu.edu

Abstract. C code is often the backbone of operating systems, and in many instances, is vulnerable to privilege escalation. CANAL is a C analysis tool written to assist in the early detection of some of these vulnerabilities. The tool is designed from multiple Python modules with a Bash script wrapper that can perform static analysis on directories of C projects. Each module looks at a different vulnerability that can be diagnosed statically via string parsing. The tool outputs warning messages to alert the end user of potential security vulnerabilities.

1 Introduction

The CANAL project processes C projects for a number of security related issues, but the prime motivation of the project was to look at programming practices that introduce privilege escalation vulnerabilities. Specifically, any programs that allow local users to perform actions with higher authorization (i.e. SUID programs) are potential targets for privilege escalation exploits. It is critical to scrutinize SUID programs to ensure they follow the principle of least privilege, and drop permissions as soon as possible. In addition, the *setuid* family of system calls, hence forth referred to simply as setuid functions, used to downgrade privilege is not uniformly implemented across platforms, and can be improperly used [1]. The CANAL static analysis tool has a number of features, but is specially crafted for identifying potentially unsafe system calls that precede privilege descalation, and for enforcing the checking of setuid return status codes.

Passing CANAL a program that calls a setuid function triggers a number of sub-module checks: the return value of the privilege deescalation family function is checked for error returns; the control flow is parsed using the *cflow*[9] tool, which is then checked for vulnerable behaviors preceding a privilege deescalation; the string or variable being passed to a system or *execve* call is run through a few naive checks to ensure the string being passed as an argument isn't one known to be unsafe; finally, there is a module that ensures the user is warned of all uses of known insecure functions such as *strcat, gets, strcpy, etc.* that might not do bounds checking or expose the program to exploitation.

The bring up of CANAL was tested on specially crafted vulnerable programs, known exploitable code[12], and university course assignments. Further analysis was then performed in mass on system utilities found in the Mac OS X[6], FreeBSD[7], and BusyBox[8] projects. Our results, including some verification of work is included in this paper.

1.1 Example Vulnerable Program & Tool Output

CANAL outputs a separate set of warnings for each type of analysis run (one for each module). Figures 1.1 and 1.2 demonstrate sample output.

Fig. 1.1 is a simple program used to illustrate what our tool looks for in terms of vulnerabilities.

```
1.  int main(int argc, char **argv) {
2.
3.    char* foo;
4.
5.    foo = (char*)malloc(200); // Should trigger "a before b"
6.    system("cat /etc/shadow"); // Should trigger "a before b"
7.    system(argv[1]); // Should trigger "check exec"
8.
9.    gid_t gid = getgid();
10.   uid_t uid = getuid();
11.
12.
13.   seteuid(uid); // Should give WARNING from "check set*"
14.   setregid(gid, gid); // Should give ERROR from "no return checker"
15.
16.
17.   char cmdbuf[128] = "export IFS=' \t\n'; nc -l 9999 -e /bin/sh";
18.    system(cmdbuf); // Should be tracked down to "export IFS=' \t\n';
    nc -l 9999 -e /bin/sh"
19.
20.   return 0;
21. }
```

Fig 1.2 Example Output from C file from Fig 1.1 - the line numbers are off due to the removal of comments in the code above:

```
[CHECK_A_BEFORE_B] Errors in: print.c
        error = malloc, line = 20, comment = [ERROR] malloc happens before seteuid on 29!
        error = malloc, line = 20, comment = [ERROR] malloc happens before setregid on 30!
        error = system, line = 21, comment = [ERROR] system happens before seteuid on 29!
        error = system, line = 21, comment = [ERROR] system happens before setregid on 30!
        error = system, line = 22, comment = [ERROR] system happens before seteuid on 29!
        error = system, line = 22, comment = [ERROR] system happens before setregid on 30!

[CHECK EXEC] Errors in: print.c
         error = system, line = 21, comment = [ERROR]  cat called in vulnerable function
         error = system, line = 22, comment = [ERROR]  argv called in vulnerable function!
         error = system, line = 38, comment = [ERROR]  error = Vulnerable input from user-accessable file in
                 37   char cmdbuf[128] = "export IFS=' \t\n'; /usr/bin/file ";

[CHECK_RETURN]print.c
        error = setgid, line = 30, comment = [ERROR] Not checking return!
        error = setgid, line = 31, comment = [ERROR] Not checking return!

[CHECK_BAD_WORDS] Errors in: print.c
        error = system, line = 21, comment = [ERROR] Dangerous exec* call!
        error = system, line = 22, comment = [ERROR] Dangerous exec* call!
        error = seteuid, line = 30, comment = [ERROR] Dangerous setuid operation. Be sure to know your OS!
        error = setregid, line = 31, comment = [ERROR] Dangerous setuid operation. Be sure to know your OS!
        error = system, line = 38, comment = [ERROR] Dangerous exec* call!
```

1.2 Motivating Vulnerability

CVE-2015-5889[14] demonstrates a recent vulnerability in *rsh,* a SUID root program. The vulnerability allows for local user privilege escalation through specifically crafted environmental variables on the Mac OS X platform for versions 10.6.8 to 10.10.5. *malloc(3)* on OS X includes features to change the behaviour of allocation related functions through environmental variables[15]. The environmental variables *MallocLogFile*, *MallocStackLogging*, and *MallocStackLoggingDirectory* can be specially crafted to edit files with the permissions of a SUID program using *malloc*. As such, *rsh*, prior to OS X 10.10.6, makes an *execv* system call to the program, *rlogin,* before dropping its root privileges. *rlogin* inherits the privileges of the program, and calls *malloc*. With maliciously set environmental variables, this execution flow allows for a local privilege escalation. This exploit illustrates the importance of deescalating privilege as soon as possible, as well as knowing the internals of your operating systems. We used this exploit as inspiration for CANAL, specifically for the **a-before-b** module. We conducted research on current methods in static analysis that would have helped with early detection of this vulnerability, and expanded to catch as many types of security issues as possible [3][5].

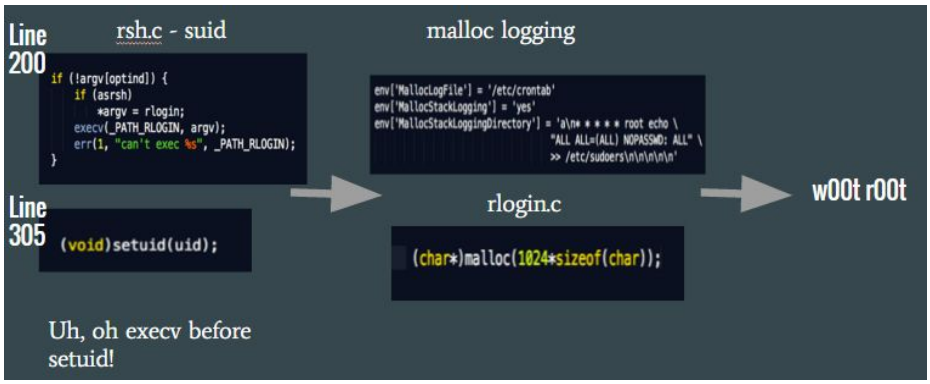

Fig 1.3 A visual demonstrating the execution flow of the motivating vulnerability that results in a root shell for any unprivileged user with access to a system running Mac OS X 10.10.5 or earlier.

2 Design Methodology & Module Overview

The static analysis tool analyzes several different aspects of C projects. The overarching utility of the tool is to check for privilege escalation vulnerabilities, and it is broken down into individual modules that each check for a specific type of error. Figure 2.1 provides an overview of the modules and overall design characteristics of CANAL's static analysis framework.

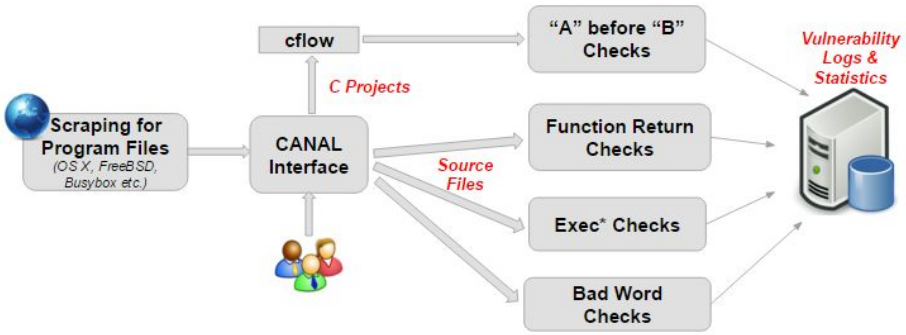

Figure 2.1 The figure above is a visual representation overview of the CANAL static analysis tool.

2.1 Pre-processing Stage

CANAL's pre-processing stage involves two main steps: (1) Scraping various popular software and OS packages from the web in an automated fashion (2) Program flow generation using *cflow*. This section details design and implementation methodology of these two steps.

2.1.1 Automated Web Scraping

In order to gain insight into the prevalence of certain vulnerabilities, the first step in our process was obtaining a sufficiently large sample of source code. This necessitated the development of an automated web scraping tool. We utilized the Python library, BeautifulSoup[16], for DOM/HTML parsing. We tailored our web scraper to identify the entirety of Apple's Mac OS X Open Source offerings[6]. We then utilize Requests[17] to download each package, approximately 20,000 packages in 90 versions of OS X.

In addition to scraping OS X packages, we also downloaded other popular software packages such as FreeBSD[7] and Busybox[8]. FreeBSD and Busybox are two widely popular software packages that power many server and embedded platforms. We then tested CANAL on these additional packages to demonstrate its applicability.

2.1.2 Program Flow Generation

In order to effectively and quickly generate program flows and determine the potential vulnerability issues that may stem from the specific order of function calls, CANAL utilizes *cflow* [9]. cflow is a freely available UNIX utility for analyzing a collection of C files to produce call graphs and flow diagrams without needing to compile the source code. As demonstrated in Figure 2.1, CANAL's main interface module generates cflow diagrams for given C projects. CANAL's main interface is implemented in bash and allows the user to specify the target C project and enable or disable various tool functionalities (e.g., submodules, program flow visualization etc.) easily by setting corresponding environmental variables.

Cflow output is provided as input for the "A before B" module. This allows the module to track the order of function calls and identify dangerous function calls that occur before dropping privileges. Figure 2.1.2 shows an example cflow output and corresponding flow diagram. As shown in this figure, cflow can effectively merge the function calls across different source files into a single call graph.

```
1   main: int (), <prog1.c 1>
2     malloc: <>
3     insecure: void (), <prog2.c 1>
4       read_passwd: <>
5       cleanup: <>
6     setuid: <>
```

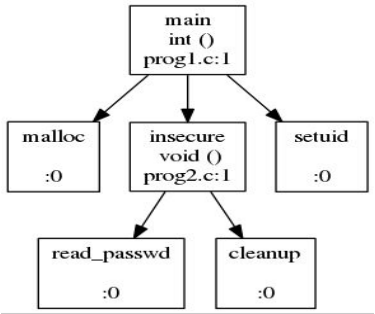

Figure 2.1.2: A sample cflow output and corresponding flow diagram. Cflow can trace and merge the function calls that occur across different source files.

2.2 Vulnerability Detection Modules

CANAL is composed of four main modules. Each of these modules reports potential vulnerabilities found in a C project.

2.2.3 Detecting Vulnerable Function Calls

This module performs a simple linear search for a list of known dangerous functions. A major limitation of this approach is the sheer number of false positives due to assuming every usage of a "dangerous" function is inherently dangerous without regard to its actual danger. Some functions on the list include functions that do not check bounds and allow buffer overflows, *gets, strcpy, strcat*. Other groups of dangerous functions include functions that allow program execution (*execve*, *system*, etc.) , and functions that allow privilege escalation (*setuid*, *setresuid*, etc.). Additionally a limitation of this module is that it only performs a string search, so commented out code, and comments that contain the the function names are detected.

2.2.4 Function Call Sequence Analyzer

This module checks for the dangerous function execution flows. We define a dangerous function execution flow as one where a function, "A", occurs before another function, "B". This flow can be seen in our motivating rsh vulnerability. Calling *malloc* before dropping privileges via *setuid* is a dangerous flow that is easily exploited. As such this module looks for occurrences of memory allocation functions before setuid functions. Additionally, this module looks for the usage of *exec* functions, before setuid functions.

The flows of programs are analyzed by utilizing *cflow* in our pre-processing step as explained in 2.1.2. By traversing the graph that *cflow* generates, we are able to determine which functions occur before others. However, this is not perfect, and due to static code analysis limitations, we are not able to follows control statements. As such, code that essentially commented out with an "#ifdef" will be considered active, and all conditional control blocks are also considered active.

2.2.5 Function Return Value Validation

This module is designed to check whether the return values of setuid functions are actually checked to verify that the setuid call successfully dropped privileges. The module performs several different checks in order to see if the return value is used for verification by the rest of the program. The first method is to see if the setuid function is called in a conditional statement. If so, the module verifies that the program performs return value check. The second method this module performs is to check if the return value of the setuid function is assigned to a variable and if that variable is used later in the program. Our module traces setuid function return values being set to a variable by traversing through the C file using string parsing and comparison.

2.2.6 Taint Checking for Exec Calls

Using unsanitized input data or program arguments in exec* or system calls introduces significant security risks due to potential injection and overflow attacks. This module is designed to address this security risk and checks whether system or exec functions are being called with arguments without proper bounds checking such as strcpy(), gets().

This module utilizes a number of string parsing functions to detect potential injection and buffer overflow vulnerabilities. Specifically, the module checks for exec* family functions or calls to command line arguments, which alias or concatenate files to which the user may have access (or be able to spoof via a symbolic link), and when functions that lack appropriate bounds checking ('gets', 'scanf', 'strcat', 'strcpy') are called as parameters. Then, the module searches the rest of the file for one level of indirection - namely variables (char * and char []) which were declared earlier in the file and initialized with these same functions and remain vulnerable to the aforementioned injection and overflow vulnerabilities.

While this module have successfully identified the target vulnerabilities in our test programs, we point to two main weaknesses. In terms of false negatives, it will fail to detect multiple levels of indirection - if a vulnerable variable is aliased to a new variable (or chain of variables) and passed, it will not be detected, and it will not trigger if the variable is the result of an unsafe function without proper bounds checking declared outside of the file. In terms of false positives, it does not have a means to detect when functions that do not have proper bounds checking are sanitized, e.g. a custom function that bounds-checks the results of strcpy(). As has been our general strategy, we err on the side of false positives. A closer approximation to the goal of covering multiple variable (re)assignment may be possible in the future using a Backus-Naur-Form specification ala pycparser[18, 19] or another grammar-based approach without heading down the path of dynamic analysis.

2.3 Post-processing Vulnerability Logs

Our final analysis includes log files for Max OS X 10.0.0, OS X 10.10.3-10.10.5, FreeBSD, and BusyBox. We generated a log of errors for each vulnerability detection module, for each of the packages we analyzed, for a total of 36 logs. Of the files we processed for statistics there are roughly ~300,000 errors reported over the ~30,000 C files. Such a high positive rate is heavily due to the vulnerable function call detection module.

3 Results and Evaluation

3.1 General Analysis & Observable Trends

Our initial thought for the statistics aggregation was to look at the first version of open source OSX released (10.0.0), and cross compare it with the most recent version available to get a feel for how coding practices have changed regarding security. For a better range of reference we compiled the chart in Fig. 2.3.1 by joining the aforementioned investigation with the results from BusyBox and FreeBSD source files.

Major Popular Package Stats:

	Mac OS X		BusyBox	FreeBSD
	10.10.5 (2014)	10.0.0 (2001)		
Total .c files:	19,372	8,440	701	1,351
Return Value Validation	0.05	0.07	0.08	0.08
Vulnerable Function Calls	2.3	2.8	2.5	2.2
Sequence Analyzer	0.4	0.25	0.9	0.14
Taint Checking for Exec Calls	0.16	0.22	0.14	0.23

Fig 3.1 Numbers in the table are given as Errors/Number of C files. Thus the number can be restated as the average number of errors per c file in a package. The reason for putting these into percentages in this manner was to make the data easier to compare on inspection across packages with large file number discrepancies.

The results cited in Figure 2.3.1 lead to a couple interesting realizations: 1) coding practices, regardless of OS, tend to exhibit very similar average errors per C file; 2) the newest version of OS X has a notable decrease in errors found through our modules compared to older versions, despite an increase in complexity of utilities.

The one area where there is an increase in the average number of errors per C file is from the sequence analyzer module. This surprising result could be the bi-product of setuid functions being used more often in more recent versions of the OS, suggesting more programs are de-escalating privileges than before. Previous programs could have just had escalated permissions from start to end, so this may be an indication of more safety rather than less. The other possibility is that the diagnostic nature of this module is more complex than most of the other modules. Due to the newer OS versions containing more complicated and multi-filed programs, we could expect more of the sequence analyzer vulnerabilities - developers may have a harder time detecting unsafe behaviours while programming across multiple files. The inspiration of the project was a critical vulnerability resulting from this very confusion.

3.2 Searching for Vulnerabilities

After the generalized post-processing we moved on to the more ambitious task of looking for an exploitable vulnerability in one of the packages, using our error dumps as the point of discovery. The mass analysis returned a large number of reports that, though they expose poor and potential insecure programming practices that could later be an issue when people modify the file without understanding previous function call hierarchy, are currently dominated by false positives.

The initial inspection we preformed involved opening the OS X 10.0 error dump and the OSX 10.10.5 error dump side by side and looking at vulnerable programs in 10.0 that had been patched in 10.10.5. This allowed us to find commonly patched outputs. Unsurprisingly we found many malloc before setuid vulnerabilities patched in the

new OS X version (for an example, see Bind v.5 compared to v.9-45) We then looked at the newest release of OS X for files that exhibited similar error patterns despite being the latest version.

Two major programs exhibited the patterns described above and were deemed appropriate for a closer look into the C program. One such program was zsh on OSX (it's almost the same name as rsh, so obviously its probably bugged right?). Though *zsh* exhibits a lot of the same insecure behaviour as the original rsh bug, it is not setuid, so with no escalated privileges to start there was no reason to de-escalate. It was odd however, that they still called a deescalation of setuid functions to user id, despite them not being issued in the first place. This finding might suggest that someone at a later date could add functionality requiring the effective id to be escalated, thus the program would allow the same vulnerability as rsh to repeat itself - that is, unless by then the developer is smart and uses *.CANAL tool in his IDE before re-deploying zsh.

Another program, or in this case a package that exhibited the same error flow as our inspiration vulnerability was Heimdal-398.40.1 in the latest OS X. Heimdal, which is a utility for Kerberos, is a complicated project with lots of nested folders filled with its own version of Kerberos'd programs. To save some time we actually were able to take a look at the cflow documents that were generated as a bi-product of our tool. At this stage we noticed that a number of the errors were spawning from a file named "rsh.c", so we had a look. As part of Heimdal's functionality it uses some adapted rsh code that it compiles from the local rsh.c - the kicker is this file has not been updated since 2007 and possess the exact vulnerability that our project was based on discovering. Now all that is left is adapting the root shell python exploit to run in Heimdal's framework and then contacting the developers through a convenient bug reporting website they have listed on their developer page.

4 Challenges

CANAL development faced several obstacles. The first challenge we ran into was web scraping. The privilege-escalation vulnerability that inspired this project is from a Mac OS X program, so Apple packages were our primary target. The consistent structure of the Apple website and discrete package availability facilitated automated web scraping. However, our goal was to obtain open source software packages for several platforms. Because website complexity and organization of packages varied across sites, Apple was the only platform for which we attained thorough web scraping coverage. Packages for BusyBox and FreeBSD were manually downloaded.

Originally, most modules of CANAL were implemented using Pycparser[18], a Python library with C file parsing functionality. Unfortunately, we discovered in early stages of testing that Pycparser cannot parse C files without detailed compilation parameters specified. Because CANAL is intended for bulk package analysis, this level of attention to individual packages was unmanageable. This forced us to restructure affected modules with native Python string-matching functions. Without the advanced features of Pycparser, we were unable to track variables passed between source packages. As a result, our taint analysis was restricted in scope. A possible resolution to this challenge is presented by the team behind Coverity[2]. They explain that through the use of Makefiles they are able to track every file that is compiled into a binary. However, they still face the limitation of code needing to be compilable, which is a non-trivial task at scale for code not maintained or owned by ourselves.

Since our tool is designed to help developers write safer code, we report all potential errors as errors. As such, there are many false positives, as we caught many cases that might actually not be unsafe but had the potential to be unsafe. Because of the amount

and variety of errors we caught, it was hard to discern which of the errors our tools caught were true errors and which were simply false positives.

5 Related Works and Future Goals

5.1 Related Works

Coverity - Static code analysis tool originating from research at Stanford by Dawson Engler and his research team. Coverity tests every line of code and potential execution paths to find security flaws in the source files [10]. In the early stages, the researchers realized that they couldn't always analyze their clients' code due to a lack of knowledge in how the code was compiled. Eventually, they realized that access to the clients' Makefiles was necessary. Now a complete success story, they were recently acquired by Synopsys for approximately $375 million.

ITS4: A Static Vulnerability Scanner for C and C++ - A static analysis tool that is a simple scanner and lacked some of the complexities our tool provides [4]. This static analysis tool could detect vulnerable functions and find simple buffer overflows, but that was the extent of the functionality. One impressive aspect of the tool was risk assessment level, the ability to discern whether an error was high risk or low risk.

Cppcheck - A free static analysis tool that has the ability to perform deep searching through multiple C files [13]. However, the major drawback is that since the recursive search is so thorough, it takes a quite a long time. Comparatively, our tool runs much faster since it does not go through so many recursive layers, and can be utilized by developers actively as they are writing programs.

Flawfinder - A free tool that does linear searches throughout a C/C++ file and reports possible security threats [11]. While it is very useful for quickly finding and removing some security flaws, it is a simple tool with very limited functionality and should not be the only static analysis tool used on a program source file.

5.2 Future Goals

IDE plugin - Successful programs are often developed into plugins for various IDEs. One possible avenue to explore is to pivot the project purpose from mass analysis towards an IDE plugin for discrete project error checking and security.

Reduce false positives - Currently, our static analysis tool raises too many false positives. While better than raising no errors at all, too many false positives certainly isn't optimal either. One possible solution to this is to have a detailed list of conditions for analysis between modules that has to be met before an error is raised. Another solution that can be implemented is to identify safe implementations of unsafe function calls so the tool returns fewer false positives.

Investigate bugs - Our tool has raised a multitude of errors from the various C files analyzed, and thoroughly investigating some of the errors raised to see whether or not any bugs can be discovered and reported is definitely something worth pursuing.

References

1. Hao Chen, David Wagner, and Drew Dean. "Setuid Demystified". In *Proceedings of the 11th Usenix Security Symposium*, August 2002.
2. A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. *"A few billion lines of code later: using static analysis to find bugs in the real world"*. *ACM Communications Magazine*, 53(2):66–75, 2010
3. Hofer T., 2010. *Evaluating Static Source Code Analysis Tools*, EPFL Thesis, InfoScience 2010. http://infoscience.epfl.ch/ record/153107
4. Viega, J., Bloch, J., Kohno, T., McGraw, G. "*ITS4: a static vulnerability scanner for C and C++ code.*" In the 16th Annual Computer Security Applications Conference (ACSAC'00), New Orleans, Louisiana, Dec 11-15, 2000.
5. Dawson Engler. "*How to find lots of bugs with system-specific static analysis"*. Stanford Lecture Notes, 2005.
6. Apple Mac OS X Source Code. http://opensource.apple.com
7. FreeBSD Source Code. https://github.com/freebsd/freebsd
8. Busybox Source Code. https://busybox.net/downloads/
9. GNU cflow. http://www.gnu.org/software/cflow/
10. Coverity. http://www.coverity.com/
11. Flawfinder. http://dwheeler.com/flawfinder/
12. Rebel . Rsh + Libmalloc Local Privilege Escalation. https://exploit-db.com/exploits/38371/
13. Cppcheck. http://cppcheck.sourceforge.net
14. CVE-2015-5889. https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-5889
15. malloc(3). https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man3/reallocf.3.html
16. BeautifulSoup. http://www.crummy.com/software/BeautifulSoup/
17. Requests. http://docs.python-requests.org/en/latest/
18. Pycparser. https://github.com/eliben/pycparser, specification at https://github.com/eliben/pycparser/blob/master/pycparser/ply/yacc.py
19. Lars Marius Garshol, *BNF and EBNF: What are they, and how do they work?* http://www.garshol.priv.no/download/text/bnf.html

Appendix

A. Roles

Name	Contribution	% of total work
Adin	Web scraping pre-processing, database system administration, post-processing collected data	16.66
Andrew	Testing, post-processing collected data, investigating vulnerabilities	16.66
Eugene	Tool evaluation, a-before-b module, overview, testing	16.66
Josh	Exec traceback module, web scraping research	16.66
Max	Return checking module, post-processing collected data	16.66
Onur	*cflow* pre-processing, testing, post-processing collected data	16.66
Jimmy John's	Provided "delicious" sandwiches	0.04

B. Source repository

Our source repository can be found on https://github.com/eugenekolo/EC521. The repository is split into several directories. The *src* directory contains the different modules, and scripts used for post and pre-processing. The *test* directory contains files

used for testing purposes, and some outputs generated during testing. The *eval* directory contains some tool evaluations we performed.

C. Favourite UNIX programs

Name	Favourite Unix Program	???
Adin	echo	echo…echo…
Andrew	scp /home/uebungen/terrier/terrier021/* . /home/thank/mr/skeletal	allows me to copy files on other ppls (eugene's exploits) accounts and change the variable names for full credit on hw
Eugene	GNU Hello	1000 lines of pure enterprise worthy joy
Josh	\(-	ascii compatible
Max	rm -rf /	secures the computer against all attacks
Onur	find	helps you find anything you need (nearby restaurants, liquor shops, bookstores etc.)
Jimmy John's	diff	a diff is essentially a sub, which is what I specialize in
Whole Group	chmod +777 terrier*	for epic chess matches on server from terminal with full animations good luck