Reinventing Bomblab

A Major Qualifying Project submitted to the faculty
of the Worcester Polytechnic Institute
in partial fulfillment of the requirements for the
Degree of Bachelor of Science

Logan Brown
Gavin Hayes
Tejas Rao

Submitted to
Professor Hugh C. Lauer,
Department of Computer Science

February 28, 2017

Acknowledgements

We would like to extend our sincere thanks to Professor Lauer for his guidance in this project. Additionally, we would like to thank Steven Malis and Fredric Silberberg for their assistance in testing the new Bomblab.

The group would also like to thank John Stegeman for his insistence that the old Bomblab could, in fact, be auto-solved without a naive solution, which led to the creation of this project in the first place.

Finally, we would like to thank Professors Professors Randal E. Bryant and David R. O'Hallaron, as well as Chris Colohan of Carnegie Mellon University for designing the most interesting and most fulfilling undergraduate assignment we have ever experienced.

Contents

Chapter 1

Introduction

Bomblab[1] is a take-home project assigned to computer science students which is intended to teach the basics of debugging and navigating assembly code. Each student is provided a randomly generated executable program referred to as a "binary bomb" which consists of six "phases" and one additional "secret phase", the latter of which is only accessible via a special key in one of the phases. This bomb is able to communicate scoring information via the Internet to a running Bomblab server specified by the instructor. The Bomblab server automatically records students' progress and maintains a public scoreboard that shows the progress of every bomb.

Each phase requires a "key" — a line of text which "disarms" the phase, allowing the student to progress to the next phase. To solve a phase, the student must reverse engineer the phase's object code to determine the correct key. This key is then entered on standard input, potentially disarming the phase. If the student enters an incorrect key, the bomb "explodes" which notifies the student of the incorrect entry and sends a message to the Bomblab server, potentially causing the student to lose points on the assignment's grade. There are three different variants for each regular phase in the original Bomblab, with only one version of the secret phase.

In their paper *Introducing Computer Systems from a Programmers Perspective*[1], Professors Randal E. Bryant and David R. OHallaron describe Bomblab as:

> ...a beautiful assignment in many ways. For the instructors, it is entirely self grading. For the students, it makes learning machine-level programming feel more like a game than a chore. It also forces students to learn to use a debugger. The *only* way to defuse a bomb is to disassemble it and then use the debugger to explore the program behavior. The bomb lab teaches students about machine language in the context they will most likely encounter in their professional lives: using a debugger to reverse engineer machine code generated by a compiler.

This posed a slight problem, as it turned out that the debugger is not the *only* way to defuse a bomb.

One of the authors, without any prior knowledge of or access to Bomblab's source code, wrote an auto-solver in April 2016 that inspects the bomb using the Linux `objdump` program as a disassembler and finds the answers to all seven phases by identifying patterns within that code and the data present in the bomb itself.[1] The existence of this auto-solver not only introduces a new way to cheat that is nearly undetectable, it also poses a significant threat to the pedagogical potential of the assignment. The fact that the solver is able to determine the keys to the bomb statically (without executing and debugging it) shows that the assignment has serious design flaws. Due to the design of Bomblab, all of the data used to check correctness is generated at compile-time and readily available to be inspected in the data sections of the executable. This makes it possible for a student to only inspect the bomb with `objdump` and avoid using the debugger altogether, violating the design intentions of the original authors.

The goal of this project is to mitigate any potential for a static auto-solver to be created. In this report, we examine the vulnerabilities of the original Bomblab, determine which options which are available to remedy these vulnerabilities, and discuss the implementation of our enhancements.

1.1 Vulnerabilities in the Original Bomblab

In the original Bomblab, bombs were generated by a Perl script that modified a set of phase templates — incomplete C source code files — and compiled the resulting code into an executable. For each phase, the script

[1]The auto-solver has not been released to the general public.

would randomly select one of three different phase templates and replace a number of placeholder values with solution data. Because all of this data is compiled directly into the executable, it is possible for the bomb to be solved without ever executing it.

The auto-solver searches for specific patterns in the disassembled machine code of a target bomb to determine the compiled variants of each phase. Once a phase variant is determined, the solver can "look up" the solution to the current phase in the executable. After the phase is "defused", the auto-solver continues on to the next phase until the bomb is completed. A brief description of the original phases and their vulnerabilities are as follows

1. In phase one, the student must identify a string stored in memory as a character array. The student's input is compared to this string, and if the two do not match, the bomb explodes. This phase of the Bomblab is trivial to solve statically, since the string is present within the executable's data section and is directly operated on with a `mov` instruction. Accordingly, not only could a student use the `strings` command to find it, he could find the offset into the executable in which the string resides, and simply read it from there. Both can be done without ever having to execute the Bomblab and risk having it explode.

2. In the second phase, the student must find a sequence of six num-

bers generated by a loop. One of three predetermined sequences is randomly selected at compile time. This phase was even simpler to solve statically than phase one as the solver does not have to read from a location in the executable; once the variant is determined, the solver inputs the predetermined solution.

3. In the third phase, the student must read a jump table generated by a switch statement in order to find both the format and the identity of the values required for the key. The structure of this phase is completely predictable, which allows a static auto-solver to assume the presence of the jump table, and simulate its operation.

4. In all but one variant of the fourth phase, the student must step through a helper function (`fun4`), which simulates searching for a node in a binary search tree. The student must enter the value of the node and the sum of the nodes traversed to reach it. However, because `fun4` accepts a finite range of inputs (integers between 0 and 14), this phase is vulnerable to brute-force solution, as noted by teaching assistants during the Spring 2016 release of Bomblab.

   The other variant of the fourth phase requires the student to enter two numbers s and b, such that $s = f(n, b)$ where n is an integer between 5 and 9 (inclusive), and f is a recursive function. Ultimately, this phase is vulnerable to auto-solution simply because n is generated at compile-time instead of runtime, and f is easily simulated.

5. The fifth phase's puzzle consists of an array that the student must identify and jump through to construct a solution. Depending on the variant, the solution may be in the form of a number or a short string. Both the array and the "starting point" of the computation are generated at compile-time, which allows an auto-solver to simulate the execution of this phase and statically determine its solution.

6. In the sixth phase, the student must identify a data structure in memory as a linked list, and determine the order in which to shuffle the elements such that the list is in sorted order (either ascending or descending). The list is always located at the same offset in the executable file, and the different variants were easily distinguishable based on the instructions thereafter. Because of this design, simulating the execution of this phase is trivial.

7. In the original secret phase, the student must determine the correct traversal of a binary search tree in order to end up at some target leaf, determined at compile-time. The tree is located at the same offset in the executable file for all bombs, and contains the same data, making this phase trivial to attack.

Chapter 2

Methodology

We determined that simplest way to mitigate potential static auto-solvers is to make the information necessary to determine the keys for each phase only available at runtime, which we accomplished by using the random number generator present in the C standard library. In order to validate submitted solutions, each bomb must still be deterministic – it must have the same solutions every time it is launched. The new Bomblab initializes the random number generator to a known state by "seeding" it, using the `srand` standard library function, with a value that is generated at compile time. Doing this forces the random number generator to produce the same sequence of numbers every time it is run. In order to defeat this type of parameter generation without using a debugger, the student would have to determine how the seed value is calculated and predict the sequence of numbers generated, in addition to Bomblab's actual challenge:

to determine how each phase makes use of those numbers. Any student capable of that kind of reverse engineering and analysis is overqualified for the lessons that Bomblab is designed to teach.

Part of our reimplementation centered around rewriting the entire Bomblab server framework in Python. The original Bomblab server was written in Perl, with development beginning during the Fall of 1998[1]. The age of the assignment and the use of Perl seriously hinder Bomblab's maintenance due to Perl's declining popularity and the fact that the Bomblab server made heavy use of custom implementations of functionality that is nowadays found in Python's (and Perl's, for that matter) standard library. In an attempt to reduce the use of non-standard Perl modules, Bomblab's code depended on custom implementations of common operations, such as date management functions. Our Python version of these scripts leverages well-tested functions present in Python's standard library. Standard library functions are maintained as the language evolves, minimizing the effort required to keep Bomblab functional in the future.

In addition to maintenance troubles, Bomblab had several performance problems that had to be resolved to ensure the success of the assignment in the future. The server consisted of four separate scripts that were each responsible for part of the functionality the assignment required. Each of these scripts ran independently of the others, which made debugging problems exceedingly difficult. In addition, each script was single threaded and thus could only process a single request at a time. This rendered it unable to generate bombs fast enough for all students at the start of a lab session. This four script system was sufficient when Bomblab was originally written, but advances in technology have rendered it inadequate for modern environments. Our rewrite is intended to make the code more readable, improve performance, and facilitate future improvements.

2.1 Server-side Bomb Production

The new Bomblab server consists of two main components: `makebomb` and `bomblab_server`. The `makebomb` script was a component of the original Bomblab design that facilitated functionality we felt was critical to preserve: It allowed the instructor to generate binary bombs from the command line with custom parameters. The `bomblabserver` script replaces the original four-script system with a single multi-threaded server. This allows the server to service numerous students' requests simultaneously and makes debugging far simpler.

The `makebomb` tool is used to generate bombs for distribution. It is designed to be invoked in two ways: manually using the command-line, or by Bomblab's server. A variety of options are exposed to the command-line interface, but only two are required: Bomblab's source directory (`src`) and the directory in which to place generated bomb distributions. All other data is retrieved from the configuration file, `bomblab.conf`, which is interpreted by a configuration parser module present by default in Python

2.7. As a plaintext configuration file, it is considerably easier to inspect and modify than the old distribution's configuration file, which took the form of a Perl module, and also segregates the script's data and behavior for more straightforward customization.

A large portion of `makebomb` is focused on evaluating command-line arguments and configuration settings to protect against invalid settings. Most options are provided as defaults or within the configuration script; when invoked using the command-line, it only requires two options: `-s`, which indicates the directory containing Bomblab's source files, and `-b`, which indicates the directory in which to generate and store bombs. The Bomblab server must also invoke `makebomb`; since it normally generates notifying bombs for use by students, it also consumes the necessary bomb ID number, the phases string (possibly blank), and the user's name and e-mail address. `makebomb` can optionally function in "quiet mode", in which all messages are redirected to a log-file or sent to `/dev/null` so that it can be operated without a terminal session.

One of our biggest priorities while developing `makebomb` was to ensure it could be used in conjunction with a multithreaded server, which would require it to be written with thread-safety in mind. The original Bomblab distribution had a single-threaded server, which was incapable of generating multiple bombs at once. This presented issues when a large group of students in a recitation or laboratory session would attempt to generate and download unique bombs all at once. Past instructors have observed that students would frequently become impatient and submit the form multiple times, bringing the single-threaded server to its knees. To fix this, `makebomb` now instantiates a `Bomb` object that can be stored in each thread, keeping all the variables involved in the bomb-making process separate. With modern servers containing dozens of CPU cores, the benefits of having multiple bombs be generated in parallel far outweigh the pitfalls.

When a new bomb is generated, the bomb ID is generated by either the server or the command-line caller of `makebomb`, before any threads are created. Other parameters, such as the user's unique, random "passkey", are generated in the thread that builds the rest of the bomb. Once these parameters are generated, a unique directory is created following the naming convention `bomb`n where n is the bomb ID, and the source directory is copied in. The purpose of creating a copy of the source files is to allow multiple bombs to be compiled simultaneously, which would not possible if the same files were used as `makebomb` modifies them during compilation. The original Bomblab used a `Makefile` for compilation which was exceedingly complex to allow for the substitutions that were required. The new Bomblab is compiled directly by `makebomb`, dramatically simplifying the process. The `gcc` compiler is invoked directly with various different parameters for the different modules of the bomb, and then the bomb is "stripped" for specific references to functions that we refer to as "black box" functions. "Black box" functions are parts of the bomb that the student is not intended to debug, such as networking and initialization code.

When a bomb is requested, several executables are generated. These include the student's bomb itself, a version called `bomb-quiet` which does not contain server code (this is used as a part of the key validation process), and a version called `bomb-solve` which outputs valid solutions to itself. In addition to the executables, the directory also contains a `PASSWORD` file, containing the randomly generated password for the bomb that is used for authentication, an `ID` file identifying the bomb, and a `README` file which identifies the bomb's user and gives basic instructions on how to use the bomb.

2.1.1 Phase Generation

Each of the first six phases consists of three source files, while the seventh phase consists of only one. When `makephases` generates a `phases.c` file, one variant (and accordingly, one `.c` file) is picked for each of the six phases. These files, along with `phase7.c` (the secret phase source file), are concatenated into a much larger `phases.c`, which is compiled into the final bomb executable.

2.1.2 Seed Generation

At compile-time, `makebomb` and `makephases` supply the bomb with two constant strings, `userid` and `user_password`. The former is (ideally) the student's username, which is required when downloading a bomb, and the latter is a unique, randomized 20-character string that consists of alphanumeric characters. The `user_password` is generated using Python's `SystemRandom` call, which on Linux employs the cryptographically secure [2] pseudorandom number generator `/dev/urandom`. These two strings — one fixed, and one random — are used to seed the random number generator. This produces enough variance in the random number generator such that the chance of two students receiving the same bomb is exceedingly low[1]. Since the random number generator is seeded with the same value every time the bomb is executed, the bomb's execution is deterministic, and the student will see the same behavior every time he or she activates the bomb.

2.1.3 Data Generation

When the bomb is executed, it calls the initialization functions for each of the six phases (called `initialize_phase1()`, and so forth). The content of these functions depends on the variant being used. In most cases, these functions consist solely of calls to the random number generator — for example, in the first phase it uses the generator to select the sentence that the user's input will be checked against. The fourth phase's initialization is more complicated, since it must generate a binary tree that appears sorted. This initialization system effectively mitigates static auto-solvers

[1]Testing of up to 500 bombs never generated two with identical solutions.

from being written, as the solution data is not present in the executable. The bomb *must* be executed for the solution data to exist, and it only exists in the executable's memory space.

2.2 Bomblab Server Architecture

2.2.1 Bomb Distribution

The original Bomblab employed two servers running on separate ports. The request server (`requestd`) handled requests for new bombs and served a static HTML file containing students' scores. The result server (`resultd`) parsed responses from bombs and logged them to a text file. Every thirty seconds a script was called that would read the log, validate every submission individually, and generate the HTML scores file. This architecture has several performance issues, the biggest of which is the fact that the entire log file is parsed for results when generating scores. This means that *every* submission will be evaluated repeatedly, with each submission requiring the execution of its corresponding bomb, even if the student has requested a new bomb to work on. In addition, because the request server was single-threaded it was not capable of generating a large number of bombs concurrently, which eventually caused the server to crash under normal class loads.

The new Bomblab uses only one multi-threaded server (as opposed to the four daemons present in the original Bomblab distribution). This server spawns a thread for each request so that they can be run in parallel. The original server could only compile one bomb at a time. In order to facilitate the compilation of multiple bombs simultaneously, we modified the build process so it copies the entire source directory into each bomb's folder before beginning compilation. This can potentially add a slight performance penalty, but it is dramatically outweighed by the ability to compile multiple bombs simultaneously.

During development of the new Bomblab server, we encountered several issues specifically related to handling parallel requests. Initially, our version of *makebomb* was essentially a direct port of the original. We changed the specific compilation commands to account for the new bomb design, but the overall flow was identical. This became a problem when we attempted to request multiple bombs simultaneously. When multiple bombs were requested simultaneously, the server would encounter a race condition and fail to build any bombs.

We then redesigned *makebomb* to create a *Bomb* object so each thread could address its bomb individually. We later ran into another race condition where two threads could both choose the same bomb ID, and fail to compile any bombs. This issue was solved by adding a python lock on the bomb ID selection routine. This way, only a single thread can select an ID at a time, eliminating the race condition.

The entire bomb validation system has also been redesigned. As before, the server accepts requests from individual assignments containing the bomb ID, the phase number, the phase's solution, and the hidden `user_password` field, which are used to verify the legitimacy of the solution[2]. However, it now validates a submission once at response time, and stores whether or not the phase was successfully disarmed (along with other student data) in a persistent score database. When the scoreboard is requested, the server is able to quickly generate a full scoreboard from the saved progress data on the fly. The new server also delivers valid W3C-standardized[3] HTML and CSS. Since bombs do not need to be executed repeatedly in order to generate the scoreboard, the Bomblab server has become significantly faster.

2.2.2 User Interface

The Bomblab server was also updated to make the client-side experience more intuitive. In order to prevent bombs with non-acceptable input text, `javascript` was used to provide feedback to the user and prevent bomb requests with invalid input text. The *submit* button can now alert the student when either the username or email is empty or the email field does not fit the format of an email. The email format is validated by by a regular expression.

In order to prevent students from generating multiple bombs at the

[2]This prevents the Bomblab from being fooled by students who, for example, jump directly to `phase_defused`.

[3]World-Wide Web Consortium

same time, in `javascript` the handler for the `onclick` event was adapted to disable to the submit button while his or hers bomb is generating. Detection of download completion was done by attaching to the `onload` event of a hidden `iframe` that performed the request.

2.3 Phases

The first three phases were modified to work with the new phase initialization system, but are essentially pedagogically equivalent. After discussion with our advisor, we concluded that two of the three variants of the fourth phase were inadequate, and would require redesign. We were also somewhat unhappy with the secret phase. The existing secret phase was far easier than phase five and phase six, allowing most students that made it to the secret phase to solve it easily. We wished for students who "discovered" the secret phase to be genuinely challenged; not everyone who found it should be able to solve it.

Instead of designing a new fourth phase and a new secret phase, we decided that the existing secret phase could be modified and used instead of phase four. The difficulty of the secret phase was on par with the original phase four, making it a suitable replacement. We then wrote a new secret phase that explored new aspects of an earlier assignment in the curriculum: Datalab.

2.3.1 The Fourth Phase

Not only was the fourth phase vulnerable to static auto-solution, variants A and B were also frequently solved using brute-force methods, defeating the entire pedagogy of the phase. On more than one occasion, teaching assistants admitted that students could use brute-force methods instead of stepping through the recursion and inspecting the call stack using `gdb`'s `back` command. The authors decided to rewrite variants A and B of phase four in order to fix this. Since phase 4C was structured differently from the first two, it was still suitable for inclusion.

The secret phase of the original Bomblab required students to find the value of a node in a predefined binary search tree given a number which represents the path taken to get to it. Just like the original fourth phase, it employed the use of a recursive function, so it was slightly modified so that every node, not just those at the bottom, has a unique value; and that the binary tree is generated at runtime.

While this now introduces operations on data structures earlier in the Bomblab assignment, the operations performed on these data structures are considerably simpler than in the sixth phase, which ought not to substantially increase the difficulty.

2.3.2 The Secret Phase

The secret phase was re-implemented for two reasons. As previously mentioned, we modified the original Bomblab's secret phase and turned it into the fourth phase of the new Bomblab. Secondly, the secret phase has become something of an open secret: in the past, the secret phase has not been referred to in Bomblab's official documentation, and instructors will often answer, "What secret phase?" when asked about it in person. Officially, the only way to find it is to find `secret_phase` in the bomb's symbol table. Despite this, rumors have flourished; when Bomblab was released to students in the fall of 2016, many students knew about it, and due to the ease (relative to the fifth and sixth phases) by which the secret phase can be solved, many people are able to both find it and solve it, which defeats the purpose of the extra phase.

Many students who complete the Bomblab do so as part of an introductory computer systems or machine organization course, modeled on a course at Carnegie Mellon University originally developed by Profs. Bryant and O'Hallaron, also the authors of the Bomblab. In the preface of the latest edition of their book [3], they strongly recommend that the internal representation of data (presented in the second chapter) be introduced to students before the analysis of machine code (presented in the third). The flagship lab assignment of the second chapter is called Datalab, in which students must "implement simple logical and arithmetic functions using a highly restrictive subset of C [3]". The specific subset varies based on the Datalab puzzle being solved, but force students to use mostly (if not exclusively) bitwise operators. The new Bomblab's secret phase was derived from the Datalab puzzle `float_f2i`, in which a student must write a function that converts the 32-bit representation (as an unsigned `int`) of a floating-point number into the actual integer it represents, rounding down to zero, and -2^{31} if the floating-point number is too big.

It is not desirable to have students rewrite `float_f2i`, especially since they may have already encountered it in Datalab, and it is impossible within the constraints of the Bomblab. Instead, they are forced to step through an implementation of `float_f2i` that uses only bitwise operations. Before the bomb starts, it generates a random integer in the range $(-1000000, 1000000)$, and expects the students' input, an integer, to match that of the 32-bit floating-point representation of that number. This secret phase is far preferable to one that introduces a new algorithm because it does not introduce any new concepts (so that all students are *capable* of doing it), but it is not so easy that a large group of students will always complete it, and its input range is not so limited that it is vulnerable to brute-force attacks.

Chapter 3

Testing

3.1 Methods

The new Bomblab was tested in two ways. We started by writing a suite of software tests to verify the correctness and performance of our code. After this first round of code-level testing was completed, an instance of the new Bomblab was made available to undergraduate Computer Science students at Worcester Polytechnic Institute. This provided insight into the stability of the server and revealed several bugs in the new server which were patched.

3.2 Software Testing

The original Bomblab server relied on functions that — in Python — would serve only to reinvent the wheel. This was due to Perl's inadequate standard library support for URL and timestamp manipulation at the time of its creation. In addition, the only test system that existed was a small script that generated a user-provided number of random bombs and ran the generated solutions file against them to verify each bomb's correctness. While this was adequate to ensure bombs were being generated and compiling correctly (and thus testing the system's installation of `gcc` and Perl, and the code within the bombs themselves), there was nothing present to validate the functionality of the server. Following best practices in software design, the new Bomblab makes considerable use of the well-tested Python standard library, instead of relying on custom written functions. This did not entirely remove the need for unit testing: there were two functions that needed to be validated: the URL lexer (`urlParser`) and the scoreboard generator (`genScores`).

`urlParser` is a wrapper around the Python standard library function `urlparse.parse_qsl` which performs additional manipulation on specific tokens. First, `urlparse.parse_qsl` is called on a request string, returning a list of arguments and values. Results strings have + symbols replaced with spaces for solution verification, and spaces are stripped from usernames as the bomb expects usernames to contain no spaces. This is validated using a series of unit tests.

`genScores` is the function responsible for reading our scores file and parsing them into an array to be presented on the scoreboard page. It contains the logic for scoring phases and deducting points for explosions. Testing for this function consisted of several unit tests that call `genScores` on a `Scores.db` file that contains a series of known scores. These unit tests check the output of `genScores` against manually calculated output.

The remainder of our codebase is not suitable for unit testing. Unit testing is suitable for functions in which a function changes an internal state in a way that can be verified. The majority of our code calls on external tools like `gcc` or the user's web browser which are out of the scope of this project. Instead, we wrote two scripts that test the external functionality of the program.

The first script, `bombsTest`, is an enhanced port of the original Bomblab test program. It calls upon functions from makebomb to generate a user-defined number of bombs and runs each against its respective solutions file. In addition to reporting how many bombs successfully compiled and ran, it checks each solution for uniqueness by hashing each solutions file using SHA-1[4] and storing the resulting hash as a string. At the end of the run, `bombsTest` counts the number of unique hashes, which represents the number of unique solutions.

In addition to testing code, we also performance tested the running server. Our initial performance tests revolved around testing scoreboard generation as that was the only component of the server we felt could have been negatively impacted by our rewrite. The original `bomblab` scoreboard was a static file that was transmitted at a student's request, whereas the new server generates the page dynamically for each request. We were unsure how many concurrent requests the server would be able to handle.

Our initial performance tests used `httperf` to send large numbers of requests for the submissions page. Unfortunately, we encountered an unforeseen issue. `httperf` is a single threaded event-based program[5] which is only able to use a single core for testing. In testing on a moderately powerful laptop, we were only able to submit between 500 and 700 requests per second before the program fully consumed the resources of one CPU core. These requests barely impacted the processor utilization of `bomblab-server`, remaining below 10 percent for the duration of the test. After some discussion, we decided that this was sufficient for the environment bomblab is targeting. An example test run can be seen in Appendix C.1.

In addition to testing the performance of the scoreboard, we had to determine if our changes to the server to facilitate multithreading had improved the performance without negatively impacting reliability. In practice, the former server would typically be unable to handle the load of an entire lab section requesting bombs within seconds of each other. Testing this once again used `httperf` to submit a lab's worth of requests. The test is designed to submit 30 bomb requests at a rate of 10 per second.

This batch of bombs took approximately 20 seconds to complete, with an average bomb compilation time of 1.5 seconds. The average connection time was approximately 16 seconds. While this is not overwhelmingly fast, it is fast enough to complete before most browsers timeout. There were no stability issues with `bomblab-server` during this test. A demonstrative set of results can be seen in Appendix C.2.

3.3 Student Testing

Through discussion with our advisor, we determined that suitable candidates for user testing are students who have already completed a previous version of Bomblab. At Worcester Polytechnic Institute, Bomblab is the second assignment in CS2011, entitled "Machine Organization and Assembly Language" [6], which is taught twice per year, mainly to first- and second-year computer science students. Computer Science students of several levels of competency were recruited for testing by sending an email to the general mailing list of computer science undergraduates at the university.

The test server was monitored closely for the first week so we could observe any potential bugs that occurred. By watching the logfile and shell output, we uncovered several bugs that we missed in testing. The first bug that was discovered was minor but made finding other bugs unnecessarily difficult. The server would log all requests to the logfile, but did not log errors, only throwing them to standard error. This was resolved by overriding the `log_message` function so that it would log any messages to the logfile in addition its previous behavior. The second bug that was discovered occurred when a student seemingly submitted a manual request to get a bomb with no parameters or credentials. While the HTML attribute `required` was applied to both the username and email fields on the form, we had not protected against manually crafted requests. We had not tested for this case, as the request form always includes the parameters even when the arguments are empty.

Unfortunately, a large number of bombs were made irrelevant during testing due to a configuration mix-up. During a patch, the configuration file was mistakenly updated to reference a laptop that was being used for development. This resulted in the bombs generated during this period being unable to submit solutions. The problem was remedied, but our pool of test subjects was severely degraded. Over the course of our testing, 70 bombs were requested. Of the 70 bombs that were delivered to students, 30 were invalidated by the aforementioned misconfiguration. Of the 40 remaining bombs, only 4 had any logged submissions. We contacted the four students that had interacted with the downloaded bombs, but none of them responded to the survey before this writing. The questions on the survey are listed in Appendix D.

Chapter 4

Conclusion

Our goal in this project was to mitigate the possible creation of static auto-solvers for Bomblab. We fulfilled this goal by redesigning the all the phases to no longer rely on statically compiled solutions. This was achieved by using a seeded random number generator to instantiate solutions deterministically at runtime. The server was rewritten in part to accommodate this change in bomb design.

Additionally, we rewrote the Bomblab server in Python for future maintainability and enhanced performance. The new server architecture is far simpler and is capable of handling workloads that crippled the original server. We replaced the original four-component server with a single multi-threaded server that takes advantage of robust standard library functionality.

Our code has been validated by extensive testing, using unit testing, black-box testing, and performance testing. Code that operated on testable data had unit tests written. Black-box testing was performed on the running server as well as the bomb compilation routine. Performance test were run on the web interface to ensure a class would not be able to crash the server.

Bomblab is a shining example of the type of engaging project work that makes the curriculum outlined in *Computer Systems: A Programmer's Perspective* [3] so successful and engaging to students. The assignment successfully makes learning the complex topic of reverse engineering software *fun*. With the changes we have made, we believe that it can continue being taught to students of computer science for years to come.

Bibliography

[1] R. Bryant and D. O'Hallaron, "Introducing Computer Systems from a Programmers Perspective," 2001.

[2] Python Software Foundation, "15.1. os Miscellaneous operating system interfaces." `https://docs.python.org/2/library/os.html#os.urandom`.

[3] R. Bryant and D. O'Hallaron, *Computer Systems: A Programmer's Perspective*. Pearson, 3rd ed., 2015.

[4] r. Donald E. Eastlake and P. E. Jones, "US Secure Hash Algorithm 1 (SHA1)." `https://tools.ietf.org/html/rfc3174`, 2001.

[5] L. Brasilino, "[httperf] multi-thread or multi-process." `http://www.hpl.hp.com/hosted/linux/mail-archives/httperf/2009-December/000617.html`, 2009.

[6] "Undergraduate Courses in Computer Science." `https://web.wpi.edu/academics/catalogs/ugrad/cscourses.html`.

Appendix A

Bomblab Description: For the Instructor

[1] Just like the old one, the new Bomblab teaches students principles of machine-level programs, as well as general debugger and very basic reverse engineering skills.

A.1 Overview

A "binary bomb" is a Linux executable program, originally written in C, that consists of six "phases". Each phase expects the student to enter a particular string on standard input. If the student enters the expected string, then that phase is "disarmed". Otherwise, the bomb "explodes"

[1] This section and the next are taken in part from the corresponding description in the original Bomblab.

(you'll know it when you see it). The goal is for the student to defuse as many phases as possible.

Each phase tests a different aspect of machine-language programs:

1. a string comparison
2. a basic loop
3. conditionals and `switch` statements
4. recursive calls and data structures
5. more complex array operations
6. sorting a linked list
7. identifying the representation of floating-point numbers.

The seventh phase is a "secret phase" that is only accessible if students append a certain string to their solutions of phases 2, 4, 5, or 6 (it depends on the student's bomb).

Each phase has three variants: 'a', 'b', and 'c'. Each student gets a bomb with a randomly chosen variant for each phase. In addition, most phase variants employ runtime-generated constants, using a seed calculated from the user's name and email address. Thus, each student gets a unique, deterministic bomb that he must solve himself. The unique solution to each bomb is available to the instructor.

In order to defuse the bomb, students must use a debugger to disassemble the binary and single-step through the machine code in each phase. The idea is to understand what each assembly-language instruction does, and then use this knowledge to infer the necessary string. Students earn points for defusing phases, and they lose points (configurable by the instructor, but typically half a point) for each explosion. Thus, they quickly learn to set breakpoints before each phase and the function that explodes the bomb. It's a great lesson and forces them to learn to use a debugger (especially now since data is generated at runtime, so the bomb cannot be analyzed using only `objdump`).

We have created a standalone auto-grading service that handles all aspects of the Bomblab for you. The Bomblab server generates new bombs on-demand, keeping one copy on the server and also packaging it in a tar file for students to download. When a phase is disarmed or the bomb explodes, the bomb notifies the same server, which keeps a log of all progress made on all bombs. The server also generates an HTML scoreboard that may be used to track students' progress.

Each time a student defuses a bomb phase or causes an explosion, the bomb sends a short HTTP message, called an "autoresult string," to the same server. It verifies that the autoresult string contains a valid solution to the phase, then updates the scoreboard accordingly.

A.2 Bomb Terminology

- **LabID**: Each instance (offering) of the lab is identified by a unique name, e.g., `d15` or `s17`, that the instructor chooses. If a bomb with a LabID different from that of the server sends it a message, the message will be ignored. It must not contain any spaces.

- **Bomb ID**: Each bomb is uniquely identified with a non-negative number. Generic bombs have a Bomb ID equal to 0, and contain no user data, and are always quiet. Custom bombs have a nonzero Bomb ID, and are always associated with a specific user.

- **Notifying Bomb**: These are compiled with the `NOTIFY` option, which causes the bomb to send a message each time the student explodes or defuses a phase. These bombs must be run on machines with specific hostnames, which is configurable in the C header file `src/config.h`.

- **Quiet Bomb**: These are compiled without the `NOTIFY` option, so the bomb doesn't send any messages to the server. These bombs can run on any host.

A.3 Offering Bomblab

There are two basic flavors of Bomblab: In the "online" version, the instructor uses the auto-grading service to handout a custom notifying bomb to each student on demand, and to automatically track their progress on the real-time scoreboard. In the "offline" version, the instructor builds, hands out, and grades the student bombs manually, without using the auto-grading service.

While both version give the students a rich experience, we recommend the online version. It is clearly the most compelling and fun for the students (as the authors will tell you from experience), and the easiest for the instructor to grade. However, it requires that you keep the auto-grading service running non-stop, because handouts, grading, and reporting occur continuously for the duration of the lab. We've made it very easy to run the service, but some instructors may be uncomfortable with this requirement and will opt instead for the offline version.

The following are instructions on how to deploy both versions of the lab.

A.3.1 Create a Bomblab Directory

A.3.2 Configure the Bomblab

Most settings can be configured by modifying `Bomblab.conf`. It's divided into two main sections: the first contains settings that you may need to modify, and the second contains settings that the instructor should only modify if you are making changes to the phases or the structure of the lab.

The following settings should be inspected before deploying the Bomblab:

- `lab-id` under **Options**, which prevents students from sending old bombs' solutions (e.g. from a previous offering) to the server.

- `explosion-penalty` under **Options**: By default, the students lose half a point for each explosion.

- `port` under **Server**: The port that the server runs on. By default, it is 15213, which corresponds to the course number at Carnegie Mellon (where the Bomblab was originally developed) in which this project is offered.

- `servername` under **Makebomb**: The hostname of the server that the bombs should report data to. This can also be an IP address, if the server has no public hostname.

A.3.3 Starting the Bomblab (Online)

To launch the Bomblab, simply run `Bomblab_server.py`. Using a Web browser, navigate to `localhost:$PORT` to test that it's running as advertised. To stop the server, simply interrupt it. The server is capable of being run in the background, but then requires a `kill` command to stop it.

A.3.4 Grading the Bomblab (Online)

The Bomblab server's scoreboard displays the score of every bomb that the server has generated. It also creates a plain text file, `Scores.db`, which contains the information in a format readable by Python's `ConfigParser`. Given the bomb number, it is simple to find the directory containing the bomb on the server, within which one can find a file containing the student's ID and email address.

A.3.5 Running the Bomblab (Offline)

In this version of the lab, you build your own quiet bombs manually and then hand them out to the students. The students work on defusing their bombs offline (i.e., independently of any auto-grading service) and then hand in their solution files to you, each of which you grade manually. You may use the `makebomb` script to build your own bombs individually; doing so will also generate the bomb's solution.

The simplest approach to offering an offline Bomblab is to build a single generic bomb that every student will attempt to defuse:

```
$ ./makebomb.py -s ./src -b .
```

This will create a generic bomb in the folder `bomb0`, as well as `bomb.c`, the main source code file (which should be distributed to students), `phases.c`, the source code for the phases (which should definitely not be), and `solution.txt`, the solution to the bomb. The students will then hand in their solution files, which you can validate by feeding to the bomb:

```
$ ./bomb0/bomb < student_solution.txt
```

This option is really easy for the instructor, but becaues there's only one bomb, it's very easy for students to cheat, since there's only one solution.

The other approach, which mimics the online version, is to build unique bombs manually for each student:

```
$ ./makebomb.py -s ./src -b ./bombs -l Bomblab -u <student's email> -v <student's id
```

This will create a quiet custom bomb in the folder `bombs/bomb<n>` for the student whose information is given on the command line. In addition to the bomb and `bomb.c`, you will also give the students `ID`, which identifies the student associated with the bomb, and `README`, which lists the bomb number, student ID, and student email. The students will then hand in their custom solution files, which you can then validate by feeding to the bomb:

```
$ ./bombs/bomb<n>/bomb < student_solution.txt
```

`makebomb` will randomly select phases using both options. If you want to generate a bomb with a specific set of options, you can pass it the `-p` option. For example, `-p abcabc` will use variant A for phase 1, B for phase 2, C for phase 3, A for phase 4, and so on.

The source code for the different phases can be found in `src/phases/` and can also be modified on demand.

A.4 Frequently Asked Questions

- **Question**: Is there any extra credit for solving the secret phase?

  Answer: What secret phase?

Appendix B

Bomblab Description: For the Student

The nefarious criminal Dr. Evil has planted a slew of “binary bombs” on my personal server. A binary bomb is a program that consists of a sequence of phases. Each phase expects you to type a particular string on standard input. If you type the correct string, then the phase is “disarmed” and the bomb proceeds to the next phase. Otherwise, the bomb “explodes”. The bomb is defused when all phases are disarmed.

There are too many bombs for me to defuse myself, so you get one. Your mission, if you choose to accept it, is to defuse your bomb before the due date. (If you don’t accept it, the consequences will be dire.) Good luck!

Step One: Get Your Bomb

Open a Web browser and go to the address your instructor gave you. You should be able to find it on your course web page. This will display a binary bomb request form for you to fill in. Enter your user name and email address and hit the Submit button. The server will build your bomb and return it to your browser in a `tar` file called `bomb`*k*`.tar`, where *k* is the unique number of your bomb.

Save the `bombk.tar` file to a (protected) directory in which you plan to do your work. Then give the command:

```
$ tar xf bombk.tar
```

This will create a directory called `./bombk` with the following files:

- `README`: Identifies the bomb and its owner – presumably you.
- `bomb`: The executable binary bomb.
- `bomb.c`: Source file with the bomb's main routine and a friendly greeting from Dr. Evil.

If you don't like the bomb you've been given, that's no big deal. Someone else will pick up the slack; you can always get a new one.

Step Two: Defuse Your Bomb

Your job for this assignment is to defuse your bomb. There are many tamper-proofing devices built into the bomb. There's also an auto-solver out there for an older version of the Bomblab – rumor has it that if you try to use it, the bomb will automatically explode!

You can use many tools to help you defuse your bomb. Please look at the **hints** section for some tips and ideas. The best way by far is to use your favorite debugger to step through the disassembled binary.

Each time your bomb explodes it notifies the Bomblab server, and you lose some number of points. Also, you could very quickly saturate the network with these messages, and cause the system administrators to revoke your computer access. So there are consequences to exploding the bomb. You must be careful!

Each phase is worth ten points of your final grade for the lab. Although phases get progressively harder to defuse, the expertise you gain as you move from phase to phase should offset this difficulty. However, the last phase will challenge even the best students, so please don't wait until the last minute to start.

The bomb ignores blank input lines. If you run your bomb with a command line argument, for example,

```
$ ./bomb solution.txt
```

then it will read the input lines from `solution.txt` until it reaches EOF (end of file), and then switch over to standard input. Despite the bomber's best efforts, I was able to add this feature so you don't have to keep retyping the solutions to phases you have already defused.

To avoid accidentally detonating the bomb, you will need to learn how to single-step through the assembly code and how to set breakpoints. You will also need to learn how to inspect both the registers and the memory states. One of the nice side-effects of doing the lab is that you will get *very* good at using a debugger. This is a crucial skill that will pay big dividends the rest of your career (providing the bomb doesn't ruin it like it did mine).

Step Three: Hand In Your Bomb

If you downloaded your bomb from a server, the bomb will keep track of your progress on it, so there may be no explicit hand-in. You can also look at your progress on the course scoreboard, which is located at `/scoreboard` under the main server. This web page is updated in real time.

Hints

- Debuggers like `gdb` allow you to view an executable's source code, or the disassembly if the source code isn't present. You may find that useful.

- Executables are usually compiled with descriptive function and variable names still present, in the form of symbols. You can see these within a competent debugger, as well as by issuing `objdump -t`. For your sake, I hope the bomber was incompetent enough to leave some symbols present.

- The TAs are your friends. So is the professor (probably). And most of all, Google is your friend.

Appendix C

Testing Results

C.1 Scoreboard Request Performance

This is a test of Scoreboard performance under extreme circumstances (1000 requests in a single second). As the results below show, the server was able to handled 515 scoreboard requests per second, although this was not a limitation of the server. The `httperf` tool was unable to generate more than that number of requests per second due to the architecture of the tool[5].

```
$ httperf --client=0/1 --server=localhost --port=15213 --uri=/scoreboard
--rate=1000 --send-buffer=4096 --recv-buffer=16384 --num-conns=1000
--num-calls=1
Maximum connect burst length: 1
```

```
Total: connections 1000 requests 1000 replies 1000 test-duration 1.940 s

Connection rate: 515.4 conn/s (1.9 ms/conn, <=118 concurrent connections)
Connection time [ms]: min 0.7 avg 123.9 max 1612.8 median 7.5 stddev 340.5
Connection time [ms]: connect 107.4
Connection length [replies/conn]: 1.000

Request rate: 515.4 req/s (1.9 ms/req)
Request size [B]: 72.0

Reply rate [replies/s]: min 0.0 avg 0.0 max 0.0 stddev 0.0 (0 samples)
Reply time [ms]: response 14.7 transfer 1.8
Reply size [B]: header 117.0 content 1927.0 footer 0.0 (total 2044.0)
Reply status: 1xx=0 2xx=1000 3xx=0 4xx=0 5xx=0

CPU time [s]: user 0.10 system 1.83 (user 5.2% system 94.3% total 99.5%)
Net I/O: 1065.0 KB/s (8.7*10^6 bps)

Errors: total 0 client-timo 0 socket-timo 0 connrefused 0 connreset 0
Errors: fd-unavail 0 addrunavail 0 ftab-full 0 other 0
```

C.2 Bomb Generation Performance

This is a test of bomb generation performance under extreme circumstances (30 bombs requested in 3 seconds). As the results below show, the server was able to handle generating approximately 1.5 bombs per second, with an average connection time of 16372.4 milliseconds.

```
$ httperf --client=0/1 --server=localhost --port=15213 --uri=/getbomb?
username=****&usermail=****&submit=Submit --rate=10 --send-buffer=4096
--recv-buffer=16384 --num-conns=30 --num-calls=1
Maximum connect burst length: 1

Total: connections 30 requests 30 replies 30 test-duration 19.943 s

Connection rate: 1.5 conn/s (664.8 ms/conn, <=30 concurrent connections)
Connection time [ms]: min 9953.7 avg 16372.4 max 18742.1 median 16806.5
stddev 1876.2
Connection time [ms]: connect 0.0
Connection length [replies/conn]: 1.000

Request rate: 1.5 req/s (664.8 ms/req)
Request size [B]: 127.0

Reply rate [replies/s]: min 0.0 avg 0.3 max 0.8 stddev 0.5 (3 samples)
Reply time [ms]: response 25.9 transfer 16346.5

Reply size [B]: header 179.0 content 71680.0 footer 0.0 (total 71859.0)

Reply status: 1xx=0 2xx=30 3xx=0 4xx=0 5xx=0


CPU time [s]: user 1.01 system 17.59 (user 5.0% system 88.2% total 93.2%)

Net I/O: 105.7 KB/s (0.9*10^6 bps)


Errors: total 0 client-timo 0 socket-timo 0 connrefused 0 connreset 0

Errors: fd-unavail 0 addrunavail 0 ftab-full 0 other 0
```

Appendix D

Survey Contents

The following survey was sent to those who had submitted at least one successful solution to the new Bomblab during our testing period.

1. What Bomb IDs did you download?
2. Generally, how difficult did you think the original and new Bomblab were, on a scale of 0 to 100?
3. How far did you get? (with options for each phase)
4. Did any phases take longer than you expected? About how long did the entire Bomblab take you?
5. Were any phases easier or harder than you expected?
6. How do you think this Bomblab compares to the original?

7. Do you think it is reasonable to expect freshmen and sophomores to complete this new Bomblab as part of CS-2011 (the Machine Organization class)?