Debugging strongly-compartmentalized distributed systems

Henry Zhu
Computer and Information Science
University of Pennsylvania
Philadelphia, United States

Nik Sultana
Computer and Information Science
University of Pennsylvania
Philadelphia, United States

Boon Thau Loo
Computer and Information Science
University of Pennsylvania
Philadelphia, United States

***Abstract*—Splitting software into distributed compartments is an important software security technique that limits the effect of vulnerabilities. Unfortunately the resulting systems are difficult to analyze or debug interactively when compared to the original. Not only are compartments distributed and executed in parallel, but they may also be strongly isolated by being sandboxed or run in minimal environments that lack debugging facilities.**

This paper is the first to study practical debugging techniques for strongly-isolated distributed compartments. We adapt ideas from other remote or distributed debugging settings to this domain, and implement and describe two radically different approaches to this problem. We evaluate these approaches both qualitatively and quantitatively, and using both toy examples and real-world open-source software. Our main finding is that out of the two approaches, using GDB remote stubs presents a good balance of performance, flexibility and usability, and we characterize this more precisely in our evaluation.

***Index Terms*—software compartments, distributed debugging, development tools**

I. Introduction

Applied to software, the *Principle of Least Privilege* (PoLP) involves minimizing the privileges that a system needs in order to function. But systems might require a large number of privileges even if only small subsets of privileges are used at different times of a system's operation. For example, at different—and possibly concurrent—phases of its lifetime, a web server needs access to the network and to different parts of the file system—to retrieve user accounts, retrieve content, and write logs—and access to back-end systems such as databases. But when handling a remote session the server usually only needs access to a subset of these—for example, the sub-system interacting with the database should not need network access.

When applied to software the granularity of PoLP is improved by *splitting* monolithic software into smaller specialised components which are called *compartments* in this paper. Each compartment is then accorded the subset of privileges that it needs to function. A compartment executes as a separate process. Further, each compartment is often *sandboxed* to seal off the compartment from system services that it does not need, and limit its access to other compartments.

As a result of this splitting, if there is an exploitable bug in a compartment's code then an adversary will be limited in what they can accomplish: they can only leverage the compartment's subset of privileges, and only operate within the sandbox. This technique has been applied to widely-used software that processes untrusted inputs, including Internet-connected servers and consumer software [12], [14].

Further, a compartmentalized system can be *strongly compartmentalized* by executing the compartments in separate machines, VMs [11], or enclaves [8], to achieve greater isolation.

The approach described so far yields more secure systems, but this comes at the expense of "in-place" interactive debugging and analysis. The resulting systems are difficult to analyze or debug interactively, especially if the compartments are strongly isolated from one another or run in minimal environments that lack debugging facilities, thus impeding the development of more secure systems. As a compromise, the system could be run as a single program during the development phase, making it amenable to standard debuggers such as GDB [10], but this approach lacks fidelity to the ultimately distributed deployment environment. For example, there may be resource- or permission-related bugs that only surface in the distributed setting. To retain fidelity with the deployment environment, we need to allow non-root and remote users to interactively debug across distributed compartments.

While distributed and remote debugging has been explored in other settings, there have been no studies on tools and techniques for debugging strongly compartmentalized systems. There is therefore a lack of understanding of which existing ideas and techniques can be borrowed into this domain, and what new issues arise in this domain.

In this paper we describe the design, implementation and evaluation of two approaches to carry out interactive debugging of compartmentalized software. Both approaches involve incorporating a small debugger into the distributed compartments themselves to service requests from a remote debugging client, but the two approaches occupy radically different points in the design space.

Our methodology consisted of first creating a *custom debugger* for distributed compartments, and implementing various standard features found in debuggers. Second, we adapted GDB's approach for remote debugging to work with the same compartments. This methodology allowed us to contrast the strengths of both approaches, including performance, flexibility and usability.

This serves as a first study of in-place interactive debugging of strongly-compartmentalized software and lays the

groundwork for follow-up research on distributed debugging of compartmentalized systems—in particular, how to retain and expand the debugger's functionality without compromising the system's security.

We evaluate both debuggers using both synthethic and real-world applications. We find that both debuggers are usable for debugging simple and real-world applications but they place different demands on the user. The main differences between the two approaches are as follows: Our custom debugger requires more source code modifications to work, compared to using the GDB approach; The custom debugger operates on a thread-based system, allowing the client to switch between debugging compartments while the GDB stub only allows a stable, single continuous connection to each compartment; Quantitative results show that the GDB stub is more efficient in the compartment's image size and its performance. The primary reason for GDB's performance advantage is that the entire state is managed by the GDB client rather than the GDB stub itself. Compared to the GDB approach, the custom approach induces greater than 70% image size overhead, and $2\times$ time overhead.

The rest of the paper is structured as follows: Section 2 describes the background and related work, Section 3 presents a tutorial-style example of debugging a common issue when working with compartmentalized software. Section 4 describes the design goals for the debuggers, and Section 5 provides more details on both approaches we implemented. Section 6 describes the evaluation of the debuggers. Section 7 discusses our results further and Section 8 concludes.

II. Background and Related Work

Compartmentalizing splits an application into compartments, where each is given a certain amount of access to code or data. This is done to prevent privilege escalation [12] and has been applied to Internet-facing software including the Chromium Web browser sandbox and OpenSSH. Compartmentalization serves to limit the attack surface available to an attacker by constraining the privileges held by each compartment, thus lessening the effects of a compromise. Existing tools for compartmentalization exist in frameworks such as PrivMan [7], SOAAP [5] and PtrSplit [9]. In addition to user-space techniques, kernel modifications to support compartmentalized applications have also been researched [3].

Typical tools for user application debugging include GDB, LLDB and WinDbg. These tools typically rely on high levels of access in order to debug a program. Usually when operating a debugger, a user requires a token which allows them to attach, read and write to a specific process. The ability to write to memory is crucial to debugging a process because, for certain architectures, breakpoints are set by overwriting instructions with a debug instruction or by setting certain processor-specific registers.

The need for remote debugging has arisen in other contexts, such as the development of embedded systems, hardware systems and large batch processing systems. These techniques are described in the next few paragraphs.

GDB remote stubs [2] have been used in related work such as bare-metal OS development and embedded systems. While GDB is a debugger commonly used to debug C code for well known and tested systems, a remote stub that implements the GDB stub protocol can used to debug processes in lesser-resourced environment.

BigDebug [6] introduces a specialized debugger meant for large scale applications. BigDebug uses a record-level tracing system to debug the applications and significantly reduce overhead from debugging. BigDebug uses simulated breakpointing which records state and allows the user to debug the application in a previous state while the application continues to run.

PhD [13] introduces a domain specific language to generate program debugging support for programs running on FPGAs. The approach works in many ways similar to the way our custom debugger works. The custom debugger requires the user to modify the source code to debug the process while the domain specific language mentioned in the paper also requires the user to write code to generate code.

There are existing debugging tools such as GDB that are able to debug compartmentalized software, but they are not convenient to use. If the compartmentalized software exists as a single process, the software can be easily debugged as any other normal piece of software would be. Nit existing traditional debuggers such as GDB are inconvenient when debugging programs with multiple compartments because their tools are not compartment-aware in the same way that they are process-aware or thread-aware. For example, a user cannot easily place breakpoints into two different compartments and view what functions were called and how data is manipulated in different compartments.

III. Usage Example

This section provides a tutorial-style presentation of how a typical program might be compartmentalized, shows a typical problem that arises, and how that problem can be debugged using both the approaches studied in this paper.

Trying to compartmentalize existing software is no easy task. Even if the compartmentalized program compiles normally after splitting the original software into compartments, there can be an enormous amount of work needed to ensure that the program behavior is not changed beyond what was desired. This is where a debugger can help. In the following section, we present an example of an attempt to compartmentalize a program using libcompart, a framework for compartmentalizing software, and walk through a common mistake people tend to make.

```
1 typedef struct Info
2 {
3   char* name;
4   char* password;
5 } Info;
6
7 // Pack structure into extension data to be
      sent from non-privilege compartment over
       to privilege compartment
```

```
8  struct extension_data
       ext_verify_login_to_arg(Info* info)
9  {
10   struct extension_data result;
11   result.bufc = sizeof(*info);
12   memcpy(result.buf, info, sizeof(*info));
13   return result;
14 }
15
16 // Pack structure into extension data to be
       sent from privilege compartment over to
       non-privilege compartment
17 Info* ext_verify_login_from_arg(struct
       extension_data data)
18 {
19   Info* result = malloc(sizeof(*result));
20   memcpy(&result, data.buf, sizeof(*result))
         ;
21   return result;
22 }
23
24 #define INFO_VERIFY_NAME "admin"
25 #define INFO_VERIFY_PASSWORD "admin"
26
27 #define LOGIN_SUCCESS 1
28 #define LOGIN_FAILED 0
29
30 // non-privileged code
31 void Login(Info* info)
32 {
33    Info info = { .name = INFO_VERIFY_NAME, .
         password = INFO_VERIFY_PASSWORD };
34    struct extension_data arg =
         ext_verify_login_to_arg(&info);
35    int logged_in = ext_int_from_arg(
         compart_call_fn(ext_verify_login, arg
         ));
36    if(logged_in == LOGIN_SUCCESS)
37    {
38       ...
39    }
40 }
41
42 // Privileged code
43 struct extension_data ext_verify_login(
       struct extension_data data)
44 {
45   Info* info = ext_verify_login_from_arg(
         data);
46   if(!strcmp(info->name, INFO_VERIFY_NAME)
         && !strcmp(info->password,
         INFO_VERIFY_PASSWORD))
47   {
48     return ext_int_to_arg(LOGIN_SUCCESS);
49   }
50   return ext_int_to_arg(LOGIN_FAILED);
51 }
```

In the example above, the program attempts to verify the user's credentials contained in the info struct (line 1) in order to successfully login to the system. The function Login() on line 31, constructs the info struct with INFO_VERIFY_NAME and INFO_VERIFY_PASSWORD, which are hard-coded correct user and password to verify with on lines 24 and 25, and marshalls the info structure into an intermediate buffer that is returned as arg to be

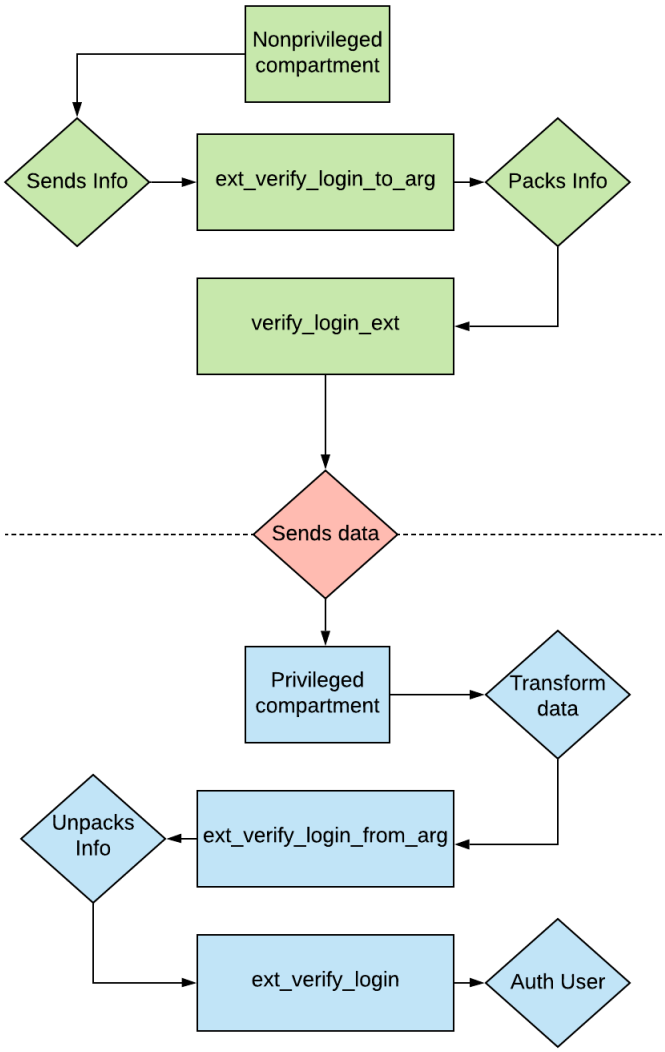

Fig. 1: CompartDebug Example

sent to the other compartment. Then, on line 35, the call to compart_call_fn tells the priviledged compartment to execute ext_verify_login along with arg, the buffer that contains the info struct, as a parameter. The privileged compartment sends back a status, packed in another extension_data struct that is unmarshalled by calling ext_int_from_arg, indicating if the user has logged into the system or not. The privileged function, ext_verify_login on line 43, verifies the user's info by comparing the user's info with INFO_VERIFY_NAME and INFO_VERIFY_PASSWORD, which are hard-coded user and passwords on line 27 and 28. Fig 1 depicits the flow.

This simple program looks like it would execute perfectly fine, but it actually results in a segfault. Using the debugger, we can figure out why.

In this example, we use the GDB stub:

```
1  // lower privileged compartment
2  (gdb) b login_interface.c:49
3  Breakpoint 1 at 0x4011a5: file login_interface.c, line
         49.
```

```
(gdb) c
Continuing.
Breakpoint 1, ext_verify_login_to_arg (info=0
        x7ffe16fd11a0) at login_interface.c:49
49      memcpy(result.buf, info, sizeof(Info));
(gdb) n
(gdb) p *info
$2 = {name = 0x404e88 "admin", password = 0x404e88 "
        admin"}
(gdb) p (char*)result.buf
$3 = 0x7ffe16fd0f68 "\210N@"
(gdb) p result.bufc
$4 = 16

// higher privileged compartment
(gdb) b login_interface.c:76
Breakpoint 1 at 0x401222: file login_interface.c, line
        76.
(gdb) c
Continuing.
Breakpoint 1, ext_verify_login_from_arg (data=<error
        reading variable: Cannot access memory at address 0
        x7ffe16fcfad0>)
    at login_interface.c:76
76      memcpy(&result, data.buf, sizeof(Info));
(gdb) n
84      return result;
(gdb) p result
$1 = (Info *) 0x404e88
(gdb) p *result
$2 = {name = 0x6f6c006e696d6461 <error reading variable
        >,
  password = 0x6e692064656767 <error: Cannot access
        memory at address 0x6e692064656767>}$
```

We begin by placing a breakpoint on ext_verify_login_to_arg, the function that transforms the info structure into an intermediate struct to be passed to the privileged compartment, after the data has been copied over into extension_data on line 2. We can see that result.buf on line 11 contains a string that does not match the user and password from the info struct on line 9. To further verify, we check the data received on the privileged compartment on line 17. When we print out the contents of the marshalled info struct on line 29, GDB catches that the name and password character pointers are indeed not valid.

We could also debug the example using the custom debugger:

```
Prompt ("stop" to quit): connect login_compartment
Prompt ("stop" to quit): list variables
Num File              Function                Line Name
0   login_interface.c ext_verify_login_to_arg 55 result
Prompt ("stop" to quit): r 0 0
{
   "name": "extension_data",
   "address": "0x7fff32eecb50",
   "val": [
      {
         "name": "bufc",
         "address": "0x7fff32eecb50",
         "type": null,
         "size": 8,
         "val": 16
      },
      {
         "name": "buf",
         "address": "0x7fff32eecb58",
         "type": null,
         "size": 512,
         "val": null
      }
   ]
}
Prompt ("stop" to quit): c
Continuing breakpoint 0...
Prompt ("stop" to quit): connect other_compartment
Connected to other_compartment
Prompt ("stop" to quit): r 0 0
{
   "name": "Info*",
   "address": "0x7fff32eeb690",
   "val": "0x405214"
}
```

We begin by connecting to the non-privileged compartment (login_compartment) on line 1. This compartment sends a request to the privileged compartment (other_compartment) to verify the info struct as shown by printing out extension_data on lines 5-25. We allow the login_compartment to continue execution, so that we can connect and inspect the privileged compartment's state on line 28. Then, on lines 30-35, we inspect the info struct, which contains an invalid pointer value that refers to the first pointer contained in the info struct in the login_compartment.

To point out the issue, the marshalling functions ext_verify_login_to_arg and ext_verify_login_from_arg are incorrect. These functions only copy over the raw data structure and thus, the data sent over is only the pointer addresses to the name and password strings rather than the characters contained in the strings. The correct version is shown as follows:

```
// Pack structure into extension data to be
    sent from non-privilege compartment over
     to privilege compartment
struct extension_data
    ext_verify_login_to_arg(Info* info)
{
  struct extension_data result;
  result.bufc = sizeof(*info);

  int current_len = 0;
  current_len = strlen(info->name) + 1;
  strcpy(result_to_arg.buf, info->name);
  strcpy(result_to_arg.buf + current_len,
      info->password);
  current_len += strlen(info->password) + 1;
  result_to_arg.bufc = current_len;

  return result;
}

// Pack structure into extension data to be
    sent from privilege compartment over to
    non-privilege compartment
Info* ext_verify_login_from_arg(struct
    extension_data data)
{
  Info* result = malloc(sizeof(*result));

  char buf[1000] = { 0 };
  int current_len = 0;
  strcpy(buf, data.buf);
  current_len = strlen(buf) + 1;
  result_from_login->name = malloc(sizeof(
      char) * current_len);
  strcpy(result_from_login->name, buf);

  strcpy(buf, data.buf + current_len);
  current_len = strlen(buf) + 1;
  result_from_login->password = malloc(
      sizeof(char) * current_len);
  strcpy(result_from_login->password, buf);
```

```
33
34   return result;
35 }
```

The correct version copies the entire string into the marshall buffer instead of copying the address of the strings into the marshall buffer.

```
1  (gdb) b login_interface.c:59
2  Breakpoint 1 at 0x401241: file login_interface.c, line
       59.
3  (gdb) c
4  Continuing.
5
6  (gdb) p (char*)result_to_arg.buf
7  $5 = 0x7ffdacd5d768 "admin"
8  (gdb) p (char*)result_to_arg.buf + 6
9  $6 = 0x7ffdacd5d76e "admin"
10
11 (gdb) b login_interface.c:86
12 Breakpoint 1 at 0x4013a4: file login_interface.c, line
       86.
13
14 // privileged
15 (gdb) b login_interface.c:86
16 Breakpoint 1 at 0x4013a4: file login_interface.c, line
       86.
17 (gdb) c
18 Continuing.
19
20 (gdb) p *result_from_login
21 $2 = {name = 0x1db22b0 "admin", password = 0x1db22d0 "
       admin"}$
```

x‘We break on the ending on ext_verify_login_to_arg and print out the contents of the buffer on lines 6 and 8. We can see here that the contents of both of the strings are successfully stored in the buffer. On the other end, the privileged compartment constructs the info struct from the buffer in ext_verify_login_from_arg. We break on the end of the function and print out the constructed info struct, which shows that both the name and password contain “admin” as a string. The info struct now contains what we expect.

IV. Design goals

Our goals in designing a compartment-friendly debugging system include providing the basic features one would expect from a modern debugger. To meet user expectations, the debugger must provide the following features to all compartments:

- Read and write to memory
- Insert and remove breakpoints
- Conditional breakpoints
- Backtracing
- Print out variables

On top of these, we provide the following:

- Query and switch between compartments.
- Debug multiple, concurrently-executing compartments simultaneously, as part of the same program.
- Communicate remotely with compartments, which might be running in separate machines or VMs.
- Debug each compartment without requiring special privileges beyond those already held by the compartment (Assuming that the compartment is reachable over the network)

A minimum set of requirements for a basic debugger consists of reading and writing to memory, stopping and continuing a program and inserting and removing breakpoints. Additionally, any modern debugger should have conditional breakpoints, backtracing, and the ability to print variables.

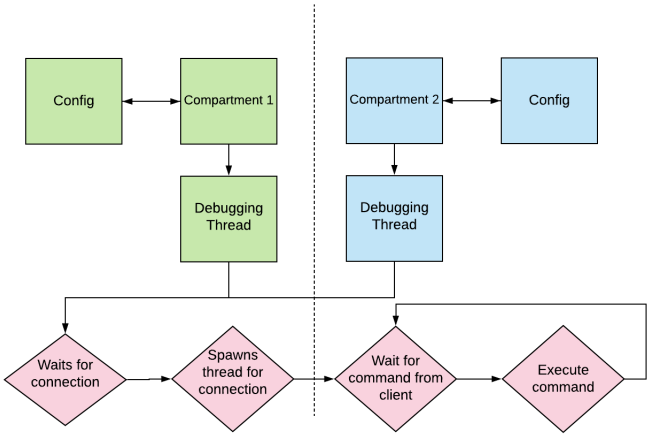

Fig. 2: Custom Debugger Server

The most important feature of our debugger is the ability to debug multiple compartments. Therefore, remote communication between the debugger and the programs’ compartments must be required because compartmentalized software does not have to run on only one machine. To achieve that goal, both of our implementations— the custom debugger and the GDB stub—use TCP for compartment communication. In addition, the debugger may not require higher privilege than the current privilege of the compartment because doing so could invalidate the compartment’s security and possibly change the control flow of the compartment.

V. Prototype debuggers

This section refines the design goals from the previous section into implementation details of both approaches studied in this paper, which will later be evaluated for their performance, flexibility and usability.

Both the custom and GDB-based approaches consist of two parts: a remote client used by the user, and a compartment-embedded server that responds to debugging-related requests from the client.

A. Custom debugger

The custom debugger is broken down into three different units, as shown by Figures 2, 3, and 4.

In Fig 2, the process that is being debugged launches the server on the compartmentalized program and eventually, a remote client connects to the server and is able to debug the process via communication to the server. The server first parses and loads a configuration file specific to a compartment, which will be explained in the next segment. Then, the server calls the SpawnDebugThread() function from the debugger’s API to create a listener thread for any clients to connect. Once a client connects, another thread is spawned to service the client’s

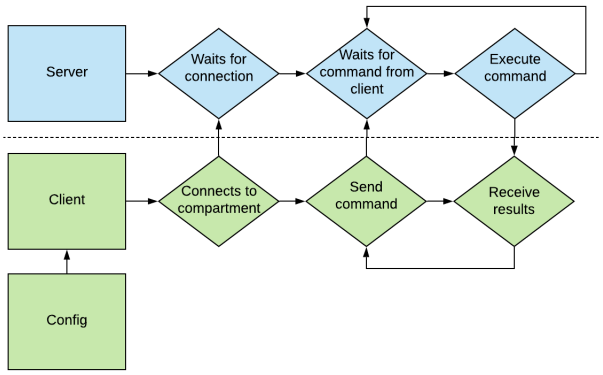

Fig. 3: Custom Debugger Client

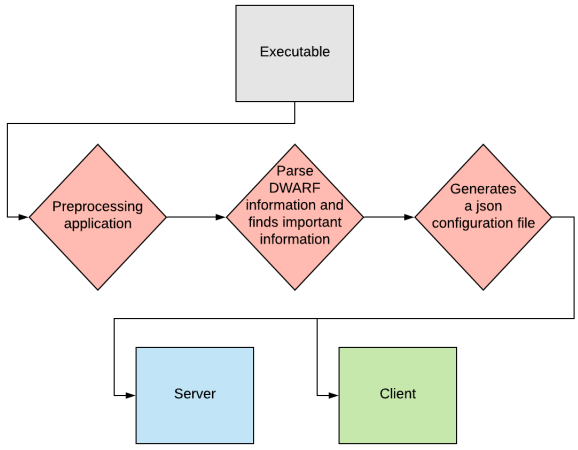

Fig. 4: Custom Debugger Configuration

requests over a TCP connection. The client sends commands over the network to be serviced by the thread.

In Fig 3, the client parses the configuration file and obtains the information about the compartments in the server. The client has to connect to a compartment. Once connected, the client can send commands to the server and the server sends back the results to the client. The client displays the results to the client.

In Fig 4, there is another preprocessing step that is required for custom debugger to work. This preprocessing step does not modify the source/binary code at all. Using DWARF [1], a standardized debugging data format debug information generated by the binary, the preprocessing step locates all the important information such as location of variables and breakpoints into the configuration file in the JSON format, which is loaded by both the server and client. Debugging the usage example under the custom debugger would require running the example through preprocessing step in order generate the debug information necessary for the debugger to operate.

Some of the features, such as breakpoints, are implemented by modifying the source code. However, actions on features such as enabling or creating a condition on a breakpoint can be done at runtime. Other features that do not modify the source code include backtracing and reading memory.

The setup process for the debugger on the server can be done in two steps:

- First, the user any compile-time feature modifications using the debugger API to insert breakpoints, load in the configuration file and indicate the compartment demarcations to the compartment-aware debugger
- Second, the user has to run the preprocessing step to generate the configuration file to be loaded by the program

In Appendix A, we provide the precise command set for this approach.

The custom debugger including the preprocessor is implemented in around 2000 lines of code.

B. GDB remote stub debugger

GDB is a widely-available debugger on UNIX-type operating systems and is a standard tool for debugging C code. A GDB stub is an implementation of the GDB protocol that allows a GDB client to communicate with the host. Usually custom GDB stubs are implemented for environments with a lack of tools to support debugging such as bare metal OSes or virtual machines. However, GDB stubs can be used for debugging compartmentalization as well. The implementation of the GDB stub is a modified version of a GDB stub implementation meant for debugging a bare-metal x86 Intel machine [4]. The GDB client features many common commands as such as breakpoints, printing variables and backtracing as well as more complex features such as scripting.

The GDB protocol uses an efficient data transfer scheme to allow the client to request information from the server. The server is required to implement the callbacks for a number of commands sent by the GDB client. As for changes to the original repository, modifications were made to support communication via TCP and handle breakpoint interrupts with signal handlers, which are handlers that can be registered to handle hardware signals. Specific amd64 assembly was used to write trap instruction that can fire a hardware breakpoint signal (‘int3’) when the instruction is executed and to read/write to specific registers.

Because the GDB stub setup is contained in one single function call, the user is only required to include the setup call per compartment.

The original GDB stub is implemented in around 1800 lines of code. The modified GDB stub supporting compartments has 400 lines of modifications to the original GDB stub (added + deleted).

C. Comparison

In Fig 5, there are some noticeable differences between the GDB stub and the custom debugger. The GDB stub has to implement architecture and platform specific code in order to support breakpoints while the custom debugger is not strongly coupled to platform and architecture. The main change necessary for the custom debugger to support other platforms is to remove thread support. The custom debugger

Feature	Custom	GDB Stub
Support on any architecture	✓	✗
Support on any platform	✓	✗
Breakpoints	✓	✓
Reading/Writing memory	✓	✓
Reading/Writing registers	✗	✓
Conditional Breakpoints	✓	✓
Scripting	✗	✓
Backtracing	✓	✓
Structure parsing	✓	✓
Source code modification	✓	✓
Hardware breakpoints	✗	✗
Source level stepping	✗	✓
Assembly level stepping	✗	✓
Send signal to process	✗	✗

Fig. 5: Feature comparison. Here *architecture* refers to the CPU the machine is running on, and *platform* refers to the OS or distribution (e.g. Windows, Ubuntu, Fedora).

requires the user to modify the source code in order to setup the debugger as well as to insert breakpoints. On the other hand, the GDB stub only requires modifications to the source code for setting up the debugger.

VI. Evaluation

Our evaluation is structured into three parts:

- a head-to-head comparison of debugger features in §VI-A,
- measurement and analysis of different overheads in §VI-B, and
- an experience report on the development and use of the two debugging approaches in §VI-C.

The snippet below is drawn from an example in the PtrSplit paper [9], consisting of a simple program that has a string format vulnerability that mimics a problem that has occurred in real deployments.

```
1  void initkey() {
2    for (int i = 0; i < 50; i++) {
3      key[i] = 'a';
4    }
5  }
6
7  void greeter (char *str) {
8    printf(str); printf(", welcome!\n");
9  }
10
11 int main (int argc, char **argv) {
12   char username[64], text[1024];
13   char *key_ptr = key;
14
15   initkey();
16
17   printf("Enter username: ");
18   fgets(username, sizeof(username), stdin);
19   greeter(username);
20   ...
21 }
```

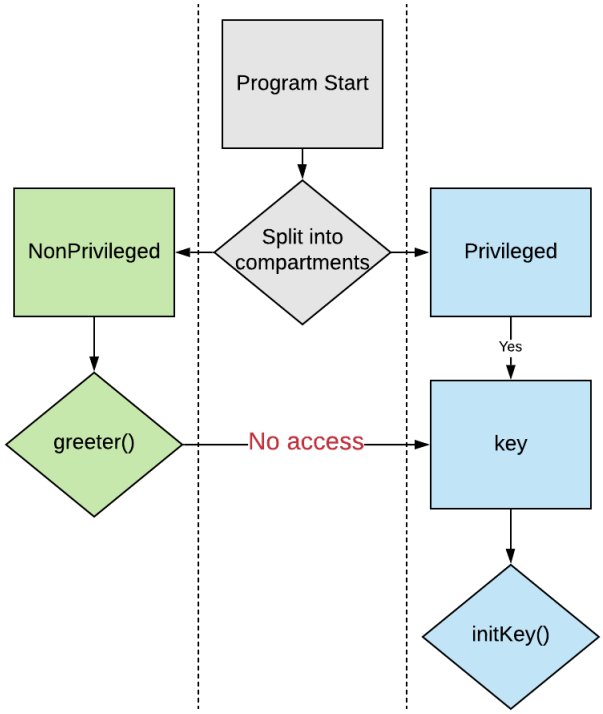

Fig. 6: PtrSplit-Fig2

In the greeter function, `printf(str)` contains the vulnerability where the user has control over the format string. Running the program normally and inputting %13$s results in printing out an offset on the stack as a string, which results in the program printing out the secret key, "aaa...". The GDB debugger can be used to observe the control flow of the program under a compartmentalized version of the program and see how the vulnerability is not leaked anymore.

In Fig 6, the program flows as follows. The program is compartmentalized into two separate programs, which are the non-privileged and privileged compartments. The privileged compartment now holds the key, which the non-privileged compartment no longer has access to. The non-privileged compartment is able to obtain the input and still has control over the string format vulnerability in the greeter function. Since the correct key is no longer in the non-privileged compartment, the key is not exposed to the user as shown by printing out the key on line 19 where the key does not have the value of "aaa...".

```
1  // GDB:
2
3  (gdb) b toy-patched.c:greeter
4  Breakpoint 1 at 0x40183f: file toy-patched.c, line 18.
5  (gdb) c
6  Continuing.
7
8  // User:
9  // Enter username: %13$s
10
11 // GDB:
12
13 (gdb) b toy-patched.c:18
14 Breakpoint 1 at toy-patched.c:18
```

```
(gdb) p str
$1 = 0x7fffb86a1a30 "%13$s\n"
(gdb) p key
$2 = '\000' <repeats 63 times>

```

The key is not exposed anymore since the actual key is contained in the privileged compartment.

So far, we have demonstrated debuggers on toy programs. A more complex example would be Netpbm, a real-world application. Netpbm is an open-source graphics program that is capable of parsing multiple types of graphics formats such as .png, and .tiff. A library under Netpbm is LibTIFF, which is a tool to read, write and manipulate .tiff files. One program in particular is tifftopnm, which converts .tiff files to another graphics format, .pnm.

```
    struct CmdlineInfo cmdline;
    ...
    struct extension_data arg;
    args_to_data_CommandLine(&arg, argc, argv
        );
    arg = compart_call_fn(
        parseCommandLine_ext, arg);
    args_from_data(&arg, &cmdline);
```

```
1703     arg = compart_call_fn(parseCommandLine_ext, arg
            );
(gdb) n

Breakpoint 1, main (argc=2, argv=0x7fff05ff15d8) at
            tifftopnm.c:1704
1704     args_from_data(&arg, &cmdline);
(gdb) p cmdline
$1 = {inputFilename = 0x7fff05ff1344 "/other/tifftopnm",
             headerdump = 3324591320,
  alphaFilename = 0x7fbdc6012000 "\177ELF\002\001\001",
             alphaStdout = 39, respectfillorder = 32701,
             byrow = 100603104,
  orientraw = 32767, verbose = 100602696}$

```

We begin by breakpointing on compart_call_fn to convert the arguments into CmdlineInfo structure. The information about the command line is sent over to the privileged compartment, which parses the arguments and converts it to a CmdlineInfo. This is obtained at args_from_data(&arg, &cmdline). When we print out cmdline, it prints out the information necessary such as the input filename that was passed to the program as well as some information about the file itself.

```
Prompt ("stop" to quit): connect netpbm_compartment
Prompt ("stop" to quit): list variables
 Num              File           Function  Line Access
                                      Name
  0     tifftopnm.c ext_verify_login_to_arg 1707 RW
                                      result
Prompt ("stop" to quit): r 0 0
{
   "name": "cmdline",
   "address": "0x7fff320b5240",
   "val": [
      {
         "name": "inputFilename",
         "address": "0x7fff05ff1344",
         "type": null,
         "size": 8,
         "val": null
      },
      {
         "name": "headerdump",
         "address": "0x7fff05ff134c",
         "type": null,
         "size": 8,
         "val": 3324591320
      },
      {
         "name": "alphaFilename",
         "address": "0x7fff05ff1354",
         "type": null,
         "size": 8,
         "val": null
      }
      ...
   ]
}

```

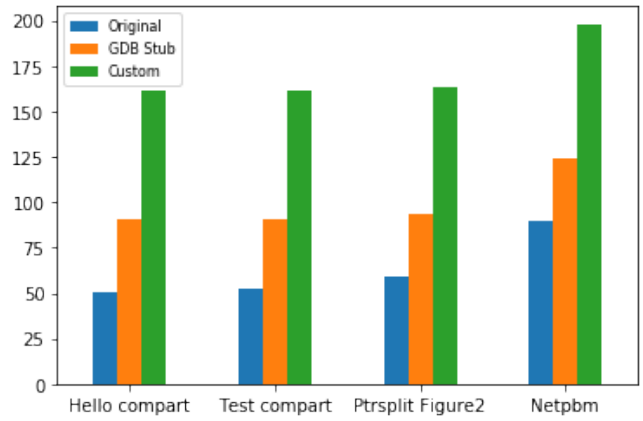

Fig. 7: Binary size (kilobytes)

Under the custom debugger, the cmdline variable can be viewed as well at the same breakpoint.

A. Features

Since the feature set of the custom debugger is mostly a subset of GDB, the GDB stub is much easier to debug with because the custom debugger requires more commands to perform same action the GDB stub is capable of in one command. There are only eleven commands that the custom uses, of which ten are a subset of GDB's commands For example, a data structure with pointers to strings or complex data structure requires the user to first print out the structure to obtain the addresses that the pointers hold. Then, the user has to cast each field that is a pointer along with its address with another command to obtain the contents at the address. On the other hand, using GDB, it is as simple as "p structure" to print out the entire structure along with subfields such as strings and pointer to other complex structures. However, the custom debugger allows the user to disconnect and reconnect to and from any compartment on a single client, compared with the GDB client where the user has to stay connected to the server and each GDB client can only select one compartment to connect to from the start.

B. System overhead

We measure the space and time overhead of each approach using several software examples.

In our results, "test compartment" refers to the example program shown in Section 3, and "hello compartment" is another simple toy compartmentalized program, which simply adds ten to an int in the privileged compartment.

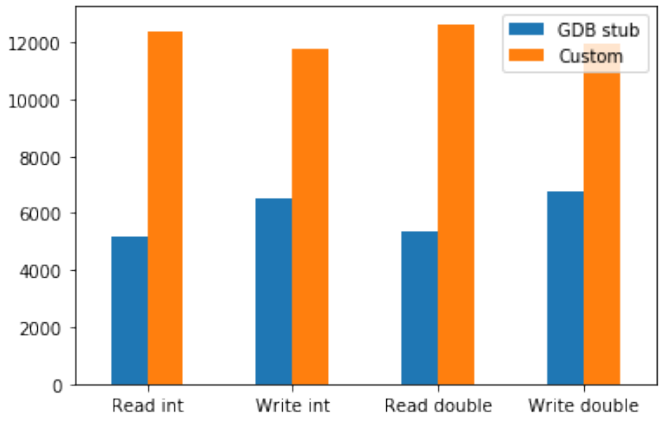

Fig. 8: Time overhead measured in CPU cycles

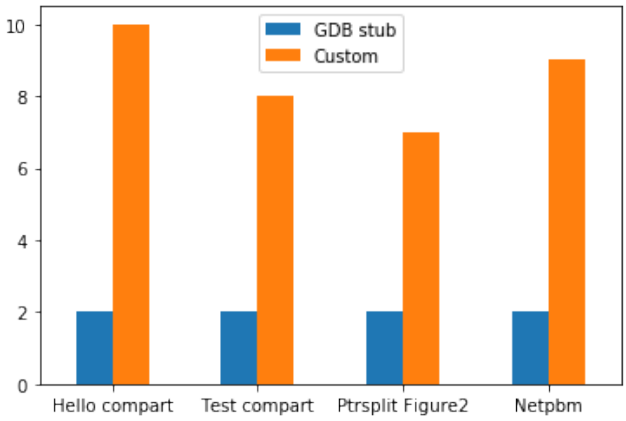

Fig. 9: Source code line changes

In Fig 7, measurements were done under linking the programs statically and compiling the program with no optimizations. The GDB stub's impact on binary size is much smaller due fact that the debug information parsing is done by the GDB client. On the other hand, the custom debugger host handles the debug parsing, which increases the binary size.

In Fig 8, measurements were done with an architecture-specific time-measuring instruction (RDTSC). The time difference is measured after the server receives information and before the server sends information to the client. The measurements were done as the difference between the start and end of the server processing the client's request. The values listed in Fig 8 are an average of ten measurements. The CPU used is an Intel Xeon E3-12xx v2 running with 3392.302 MHz and runs Ubuntu 16.04.5 LTS under Linux kernel version 4.4.0-137-generic. The GDB stub is faster because data sent from the client is more compact and easier to parse than the data to parse in the custom debugger.

Key quantitative results from the Fig 7 show that the custom debugger and GDB stub incur 171% and 71% binary size increase respectively. In Fig 8, the custom debugger has approximately a 2.05× overhead on average for reading and writing primitives compared to the GDB stub. The GDB stub transfers exactly 8 bytes for reading primitive while the custom debugger transfers 20 bytes on a 64-bit machine. On average, the custom debugger requires more than twice the amount of source code line changes for setup. In total, the source code line changes average around a 325% increase when using the custom debugger.

The GDB stub is much more efficient in binary size and performance than custom debugger is. The GDB stub does not keep state on server and uses an efficient protocol to transfer data to and from the client. On the other hand, the custom debugger manages all the state on the server and uses a simple, but inefficient protocol to transfer data to and from the client and server, which incurs an penalty in creating data that matches the protocol. This results in a bigger binary size and a performance penalty hit for the custom debugger.

C. Experience report

Implementing and using the approaches described in this paper taught us about what kinds of problems seem to surface more frequently when debugging compartmentalized systems. Setting up the system for debugging seems especially prone to problems. For example, there was an instance where the GDB client was able to debug compartmentalized software using the wrong symbols. There were two versions of the executable file where one file was older than other. The GDB client was not able to pick up on the differences and continued debugging with information that did not match the source code. Setup with the custom debugger can be extremely tricky at times and missing a certain step can lead to time lost. The key transformations for the custom debugger are:

1) transform the source code correctly to setup the compartments,
2) run the preprocessor code to generate the configuration for the debugger to use, and
3) make sure the configuration file generated has compartments that match the compartments in the source code.

If any of the steps is not performed correctly, finding the error can be extremely difficult because it is hard to pinpoint where the issue is. There were many cases where we tried to figure out an error and had to meticulously go through each of the above steps in order to pinpoint the problem. In most cases, this did not point to fundamental problems of an approach, but rather the prototypical nature of the tools.

VII. Discussion

Writing code can always introduce bugs. The debugging tools themselves can introduce new bugs that can compromise the security of a compartment. By locking down the compartments when debugging, we can avoid sacrificing security for more convenient debuggability. In both our implementations, the debugging instances has the same level of privilege as the compartment it is attached to.

In our prototype, we do not protect the debugging interface or connection—for instance, the information exchanged between the debug client and server is not encrypted and can

be manipulated by third parties. Further research is needed to secure this interface without impoverishing its functionality.

In addition, better usability is another important practical consideration. Currently, the setup for the custom debugger is much more complicated than the setup for the GDB stub. Automated code transformation of the source code would help generate the necessary setup function for the custom debugger.

VIII. Conclusion

There is a lack of debugging tools for compartmentalized software. Traditional debuggers such as GDB and LLDB are not suited to debug compartments since compartments are interfaced differently from programs that the traditional debuggers are designed for. This paper presents and evaluates two approaches to address that gap. Both approaches can be used on *strongly compartmentalized* systems, in which different compartments execute on separate machines or VMs. These approaches provide the necessary debugging primitives to inspect memory and to pause the compartments, as well as more advanced features such as the ability to inspect data structures and create conditional breakpoints. These approaches occupy different points in the design space, and our evaluation showed their usability and performance characteristics.

Acknowledgment

We thank Zhilei Zheng for reproducing the example from the PtrSplit paper [9] which we developed further for the evaluation in Section VI. We also thank Zhilei and the anonymous referees for their helpful feedback on this work. This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contracts No. HR0011-19-C-0106, HR0011-17-C-0047, and HR0011-16-C-0056. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA.

References

[1] "DWARF Debugging Information Format Version 5," accessed December 2019. [Online]. Available: http://www.dwarfstd.org/doc/DWARF5.pdf

[2] "(gdb manual) Debugging Remote Programs," accessed December 2019. [Online]. Available: https://sourceware.org/gdb/current/onlinedocs/gdb/Remote-Debugging.html

[3] A. Bittau, P. Marchenko, M. Handley, and B. Karp, "Wedge: Splitting Applications into Reduced-Privilege Compartments," in *NSDI*, vol. 8, 2008, pp. 309–322.

[4] M. Borgerson, "gdbstub," https://github.com/mborgerson/gdbstub, 2020.

[5] K. Gudka, R. N. Watson, J. Anderson, D. Chisnall, B. Davis, B. Laurie, I. Marinos, P. G. Neumann, and A. Richardson, "Clean Application Compartmentalization with SOAAP," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: ACM, 2015, pp. 1016–1031. [Online]. Available: http://doi.acm.org/10.1145/2810103.2813611

[6] M. A. Gulzar, M. Interlandi, S. Yoo, S. D. Tetali, T. Condie, T. Millstein, and M. Kim, "BigDebug: Debugging Primitives for Interactive Big Data Processing in Spark," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 784–795. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884813

[7] D. Kilpatrick, "Privman: A Library for Partitioning Applications," in *USENIX Annual Technical Conference, FREENIX Track*, 2003, pp. 273–284.

[8] J. Lind, C. Priebe, D. Muthukumaran, D. O'Keeffe, P.-L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eyers, R. Kapitza, C. Fetzer, and P. Pietzuch, "Glamdring: Automatic Application Partitioning for Intel SGX," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, Jul. 2017, pp. 285–298. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/lind

[9] S. Liu, G. Tan, and T. Jaeger, "Ptrsplit: Supporting general pointers in automatic program partitioning," in *CCS '17*, 2017.

[10] M. Loukides and A. Oram, "Getting to know gdb," *Linux J.*, vol. 1996, no. 29es, Sep. 1996. [Online]. Available: http://dl.acm.org/citation.cfm?id=326350.326355

[11] D. G. Murray and S. Hand, "Privilege Separation Made Easy: Trusting Small Libraries Not Big Processes," in *Proceedings of the 1st European Workshop on System Security*, ser. EUROSEC '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 40–46. [Online]. Available: https://doi.org/10.1145/1355284.1355292

[12] N. Provos, M. Friedl, and P. Honeyman, "Preventing Privilege Escalation," in *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, ser. SSYM'03. USA: USENIX Association, 2003, p. 16.

[13] N. Sultana, S. Galea, D. Greaves, M. Wójcik, N. Zilberman, R. G. Clegg, L. Mai, R. Mortier, P. R. Pietzuch, J. Crowcroft, and A. W. Moore, "Extending programs with debug-related features, with application to hardware development," *CoRR*, vol. abs/1705.09902, 2017. [Online]. Available: http://arxiv.org/abs/1705.09902

[14] R. N. M. Watson, J. Anderson, B. Laurie, and K. Kennaway, "A Taste of Capsicum: Practical Capabilities for UNIX," *Commun. ACM*, vol. 55, no. 3, p. 97–104, Mar. 2012. [Online]. Available: https://doi.org/10.1145/2093548.2093572

Appendix

Command	Meaning
connect compartment_name	connect to compartment with the name compartment_name
info b	print information about the breakpoint
bd n	disable the breakpoint 'n' where n is the index of the breakpoint from info b
be n	enable the breakpoint 'n' where n is the index of the breakpoint from info b
c	continue execution
list variables	list the variables at the current breakpoint
r n v	read the variable with the index 'v' at breakpoint index 'n'
rr (type)address	read a structure of type at address. Ex. rr (int)0x8000 reads an int at address 0x8000
w (type)address value	writes a primitive type of value at address Ex w (int)0x800000 100 writes an int containing 100 at address 0x800000
bc n v comparison value	make a breakpoint at index n, checking value against the variable's value at variable index v with the comparison. Comparison can be of ['!=', '==', '<', '>', '≤', '≥']. Ex. bc 0 0 == 100

TABLE I: Custom debugger commands