Revitalizing DCE

[Extended Abstract] *

AuthorName
Institute for Clarity in Documentation
mail@mail.com

ABSTRACT

This paper describes the design, implementation and validation of the ns-3 model of the Licklider Transmission Protocol, the standard transport protocol used to provide transmission reliability in Delay Tolerant Networks (DTNs). DTNs are an emerging field whose principles are used to provide communications in extreme and performance-challenged environments, such as spacrecraft,underwater, or disaster response scenarios. Evaluation of such environments requires the use of simulation tools. As of now, there is a lack of precise simulation models of these protocols, and concretely within the ns-3 simulator. The ns-3 model presented in this paper accurately models the LTP protocol and offers ...

Categories and Subject Descriptors

C.2.2 [**Computer-Communication Networks**]: Network Protocols—*Protocol architecture*; I.6.5 [**Simulation and Modeling**]: Model Development

General Terms

Theory

Keywords

ACM proceedings, LaTeX, text tagging

1. INTRODUCTION

The rest of this paper is organized as follows: Section 2 provides an overview on Delay Tolerant Networks and its transmission protocol standards. Sections 3 describes the design and implementation of the Licklider Transmission Protocol ns-3 module. Section 4 presents the testing approach procedure. Section 5 shows the validation procedure for achieving interoperability against existing implementations. Lastly, section 6 offers the conclusions and future work.

2. CHALLENGES

2.1 libio vtable mangling

The vtable is abstractly a table maintainig references to functions called for virtual functions defined for a class or an entity. These functions can be overriden dynamically by user defined functions and the respective call for the coorresponding virtual function in a derived clas object can be bound to that function at runtime, unlike pre-defined functions which are static and fixed, and can be determined during compile time. The libc on Linux provides this highly flexible feature for all other user defined classes but the case with the FILE structure is not the same.

The FILE structure is a library defined structure which defines the overall organization, orientation and properties of any file I/O stream opened by the host application. It maintains different parameters to store useful operational fields like the UNIX based file descriptor number of the opened stream, the read/write offsets and buffer addresses of the stream. The pseudoname for the FILE structure as seen inside libc is *_IO_FILE*. Since, FILE is a library defined entity, the library provides it's own set of implementation for all possible operations on an open FILE stream. Whenever an *_IO_FILE* stream is allocated by the kernel, a contiguous memory location is allocated as a block called *_IO_FILE_plus*. The *_IO_FILE_plus* structure looks like this.

```
1 struct _IO_FILE_plus
2 {
3   FILE file;
4   const struct _IO_jump_t *vtable;
5 };
```

Now, by nature of implementation of the kernel's memory allocation processes, the contents of a struct are allocated in contiguous memoory locations. This can be veriied by the *sizeof* operation of C to verify that the sum of sizes of the individual members of a struct is equal to the size of the struct object. Similarly, the FILE and the *_IO_jump_t* objects are allocated in contiguous memory locations. Specifically, the *_IO_jump_t* areas is interesting to us, as it defines the callbacks or refference pointers to the functions handlers for each supported file operation. Some of the callback which are interesting to DCE and it's use cases are highlighed below.

*A full version of this paper is available as *Author's Guide to Preparing ACM SIG Proceedings Using LaTeX2ε and BibTeX* at `www.acm.org/eaddress.htm`

WOODSTOCK '97 El Paso, Texas USA

```
1  struct _IO_jump_t
2  {
3  ...
4  ssize_t(*) __read (FILE *, void *, ssize_t)
       ;
5  ssize_t(*) __write (FILE *, const void *,
       ssize_t);
6  off64_t (*) __seek (FILE *, off64_t, int);
7  int (*) __close (FILE *);
8  int (*) __stat (FILE *, void *);
9  ...
10 };
```

This structure acts like the vtable for the FILE structure, but it does not behave like the oridinary vtable seen when working with virtual functions and derived classes, which are dynamic and supports run time bindings. This vtable is rather expected to behave as a statically bound vtable (there does exist other libc functions like *fopencookie*, to override some of the FILE operation implementations, but not all, and it also does not atach itself to a standard file I/O stream, and rather works with user defined buffers a.k.a cookies).

DCE, which is supports simulating real applications on top of the ns-3 stack and also provides different networking stack choices (Linux, ns-3, FreeBSD), for the host application being run to get a simulated real world execution environenment and results. To support such an implementation and to sync application operations like system calls, file I/O operations, networking system calls, etc. it need to hijack all such calls and re-route it through corresponding handlers based on the application logic and simulation script implementation. Considering file I/O operations, DCE needs to have control over read/write/close/seek/stat operations of each open file, which requires us to overwrite the vtable handlers with the corresponding handlers defined in DCE's stdio definition source files.

Taking advantage of the contiguous memory allocation of the FILE and *_IO_jump_t*, we can execute a buffer overflow attack on the FILE object to overwrite the vtable with our custom vtable definition for all the operation we would want to overwrite. We can make a dummy *_IO_FILE_plus* pointer point to the explicitly casted FILE object. *memcpy* the existing vtable to a local copy, modify and overwrite the stream operations with our custom written implementation, and then re-point the vtable field of our dummy *_IO_FILE_plus* to the local modified vtable, and now we have control over those stream operations, which can now be routed through and and to behave as ns-3 streams, Unix FD streams, etc. based on the type of file descriptor that is defined. This is one of the productive uses of an buffer overflow attack to leverage control over FILE streams to regulate stream buffer flushing and data redirection, but the same could be used for use cases which might pose as potential security threats, as it lets penetration testers to make use of tools like pwntools etc. to gain control over application execution and important run time CPU register values such as the *rip* which is used for the relative addressing of application components(which is also how position-independent-executables work), which is more secure as compared to static addressing, where fixed address values of symbols and pointers could be gained by static analysis tools for run time application exploitations.

Post libc-2.25, security features have been implemented to glibc to identify such buffer overflow attack. Whenever any FILE operation is executed, glibc would verify if the FILE object's vtable could be trusted and is not corruputed or manipulated. To verify this, it makes a call to *_IO_validate_vtable*. Every libio vtable is defined in a unique section called *libio_IO_vtables*. By definition, libc would trust the vtable if the vtable of the current FILE object lies within this section. It checks if the offsets of this vtable lies between *__stop__libc_IO_vtables* and *__start__libc_IO_vtables*, if it does, we can continue with the operation, if not, libc conducts a final check by calling *_IO_vtable_check* which makes final checks on the FILE vtable pointer location, namespace and edge cases where FILE * objects are passed to threads which are not in the currently linked executable. Since, when we oveflow the *_IO_FILE_plus* and overwrite the *_IO_jump_t*, it does not lie in *libio_IO_vtables* section and it also does not pass the pointer mangling sanity checks, leading to a *__libc_fatal (Fatal error: glibc detected an invalid stdio handle);*

2.2 PIE loading and usage

PIE or position-independent-executables are applications compiled with special compiler flags, which allow the application to be loaded into random memory address, not depending on absolute symbol addresses, avoiding exploits which hijack the call stack by referencing constant function/symbol addresses. In the case of a PIE, every memory address is accessed with refference to what is called the *%rip*. The *%rip* is computed at the time of execution when the application is loaded into virtual memory. This makes it difficult for attackers to determine symbol location in memory.

In DCE, we support the execution of real host application in simulation, bridging the networking layer between the host and the specified networking stack, and also other system calls made by the host applications. Since, we might have to load several applications into memory, and alsp have control over the position of the main symbol of the loaded application, we need to have position independent executables, so that when they are loaded into memory, the symbol positions in memory are dynamic, giving us control over when an application is launched in a simulation which can be configured in the script using available ns-3 programming constructs for the *DceApplicationManager* class. To implement this, DCE used the *CoojaLoader* which uses dlmopen udner the hood to load the executable into memory. In glibc version newer than 2.25, security checks have been introduced to identify such PIE objects being loaded through dlmopen, and in case it finds the **DF_1_PIE** flag in it's ELF Dynamic headers, if would abort with an error.

2.3 Linux Networking Stack for DCE (LKL vs LibOS)

Talk about other DTN simulators/modules: the ONE [3] , omnet++ [2], there are also some modules for ns-2 (not totally sure).

This subsection may be worth moving to the introduction as just a simple paragraph

3. SOLUTIONS

3.1 Custom glibc-2.31 Based Build

3.2 Bake Build Automation

3.3 Docker environenment for DCE

3.4 net-next-nuse-5.10

Communication between the LTP engine and the Client Service Instance can happen both ways. The client service makes requests to the LTP engine (start or cancel transmission) and the LTP engine issues back notifications (to report certain events or hand over received data). The design of these communications is a local implementation matter, in the ns-3 module:

- Requests are provided in the form of API functions ...
- Notifications are provided as callback functions ...

The LTP may be run at different protocol layers in order to provide support for this we provide Convergence Layer adapters ...

LTP uses engine IDs as its addressing system, we provide a lookup structure in the form of LTP to IP resolution tables ...

4. RESULTS

4.1 Docker vs Native DCE Simulation Tests

4.2 Performance : DCE vs. ns-3

4.3 Google BBR v1 Validation Results

5. RELATED WORK

6. CONCLUSIONS

7. ACKNOWLEDGMENTS

8. REFERENCES

[1] Space networks user's guide (snug). Technical Report Revision 9, Goddard Space Flight Center, Greenbelt, Maryland, August 2007.

[2] O. R. Helgason and K. V. Jónsson. Opportunistic networking in omnet++. In *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*, Simutools '08, pages 82:1–82:8, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[3] A. Keränen, J. Ott, and T. Kärkkäinen. The one simulator for dtn protocol evaluation. In *Proceedings of the 2Nd International Conference on Simulation Tools and Techniques*, Simutools '09, pages 55:1–55:10, ICST, Brussels, Belgium, Belgium, 2009. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[4] Ltplib documentation and test. http://dtn.dsg.cs.tcd.ie/sft/ltplib/.

[5] Ltp-ri specification and implementation. http://irg.cs.ohiou.edu/ocp/ltp.html.

[6] The interplanetary overlay network (ion). http://ion-dtn.sourceforge.net/.

[7] Common open research emulator (core). http://www.nrl.navy.mil/itd/ncs/products/core.

[8] R. Wang, S. C. Burleigh, P. Parikh, C.-J. Lin, and B. Sun. Licklider transmission protocol (ltp)-based dtn for cislunar communications. *IEEE/ACM Trans. Netw.*, 19(2):359–368, Apr. 2011.