vlogin: A Prototype on Counterfeit Object-Oriented Programming and Data-Only Attacks

Simon Greenblatt
Brown University

Abstract

The wide-spread use of ASLR, shadow stacks, and CFI have made code reuse attacks more difficult. However, data-only attacks, which target non-control data, are able to bypass many of these defense mechanisms as they don't rely on hijacking a program's control flow. Though very application-specific, these types of attacks are powerful as they operate on the natural execution flow of a program to achieve an objective. Counterfeit Object-Oriented Programming (COOP) is a subset of data-only attacks that relies on the modularity of inheritance and polymorphism in C++ programs by injecting counterfeit objects into memory that invoke virtual functions that are allowed by the CFG. In this paper I introduce vlogin: a vulnerable program that demonstrates the principles of COOP and privilege escalation by performing an administrator-level action from within a guest-level domain. I explain the design and vulnerabilities of vlogin and walk through the reverse-engineering process as well as the attack approach. Overall, vlogin is a simple prototype that showcases the steps needed to carry out a data-only attack.

1 Introduction

Software defense and attacks have been in a decades-long arms race of increasing complexity. Traditional code injection utilizes a memory corruption vulnerability to write shellcode into memory and then hijacks control flow to execute it. These attacks can be easily defended against by using Data Execution Prevention (DEP) or by marking memory pages using a W$\oplus$E policy. In response to these defenses, attackers came up with code-reuse attacks such as Return-Oriented Programming (ROP) and ret2libc. These attacks reuse snippets of code known as gadgets to stitch together a payload. However, much work has been done to mitigate ROP attacks including Address Space Layout Randomization (ASLR), shadow stacks, and Control Flow Integrity (CFI) [1,2,5]. These defense mechanisms make it much more difficult to locate gadgets, hijack control flow, and execute code that doesn't adhere to a program's Control Flow Graph (CFG).

A new attack paradigm has emerged in recent years that doesn't rely on hijacking control flow but rather targets non-control data. These data-only attacks are powerful since they operate on the natural execution flow of a program to achieve an objective and can't be mitigated using CFI. By modifying local/global variables, data structures, or data pointers, it's even possible to perform an arbitrary Turing-complete computation.

1.1 Counterfeit Object-Oriented Programming

Counterfeit Object-Oriented Programming (COOP) [6] is a subset of data-only attacks that relies on the modularity and inheritance of polymorphism in C++ programs by injecting counterfeit objects into memory that invoke virtual functions that are allowed by the CFG. In order to understand how a COOP attack works, it's important to know how inheritance and polymorphism work behind the scenes in C++.

Objects in C++ can be represented as classes which typically store functions and variables related to the object. Some classes may share many functions and variables with other classes as they may be part of a hierarchy. C++ provides a convenient way to reuse data and code across multiple related classes so as to better encapsulate the hierarchical relationship between them. With *inheritance*, a parent class defines the functions and variables that child classes will derive. Child classes may then provide additional functions and variables that are specific to their needs. However, some functions that are defined by the parent class and that are common to all child classes might need different implementations for each child class. This can be accomplished using *virtual functions* which can be overridden and dispatched dynamically. Known as *polymorphism*, this technique allows for child classes to provide their own custom implementation to shared functions. For example, a Vehicle class might define functions and variables relevant to all vehicle types. These might include a fuelAmount variable or a applyThrottle() virtual function. Child classes that inherit from the Vehicle class, perhaps a

Bus or Car class, will provide a concrete implementation of the applyThrottle() function in a way that is relevant to their particular needs.

Function pointers to concrete implementations of virtual functions are stored in arrays known as *vtables*. Each child class that inherits from a parent class has its own vtable which stores pointers to the virtual functions defined in the child class. In fact, when a child class is declared, the first thing that is stored in memory is a pointer to its vtable. This is then followed by any data or other functions defined in the child class. The key observation here is that there is some ambiguity as to which vtables are allowed for each child class. The modularity of inheritance means that an attacker could craft a counterfeit object whose vtable pointer points to that of a different child class that also inherits from the same parent class. By injecting these sorts of objects into memory and connecting together chains of virtual functions, it's possible to perform a malicious computation.

Two problems remain, however, with this approach: how to control the order in which virtual functions are dispatched and how to pass relevant arguments between functions. The former problem can be addressed with *dispatcher functions* [4] which iterate over several objects and invoke a virtual function on each object. This is a rather common pattern in object-oriented languages as one could imagine a data structure's destructor function calling the virtual destructors of each of its member nodes. If these member nodes are counterfeit objects injected by an attacker, then she may be able to tamper with their corresponding vtables and thus control the order and type of virtual functions that are invoked. The later problem can be addressed by overlaying objects in memory such that fields from different classes point to the same location in memory. This allows for a function to act on a data field and then for another function from a different overlaid object to act on the same field, thus allowing the same data to flow across several functions.

2 Design

vlogin is a simple C++ program that allows a user to create a guest object, print the names of all the users in the system, and perform an action. This action is nothing more than printing the privilege level of the user on the screen. There are two types of users: guests and administrators. Guests can perform guest-level actions while administrators can perform administrator-level actions. Internally, these classes and functions are implemented using inheritance, polymorphism, and virtual functions. The User parent class defines a name field and the printName() and performAction() virtual functions. Both the Admin and the Guest child classes inherit from the User class and provide concrete implementations for both of these virtual functions. The UserList class stores the state of the program including an array of User object pointers and the index of the current user. The currentUser variable is referenced in the userList array when calling the performAction() function. The main() function creates three guests and one administrator, takes the user's input for a guest's username, creates the guest object, and contains the main loop of commands. In its current state, a user can only create a Guest object and there is no way of creating an Administrator object or performing an administrator-level action.

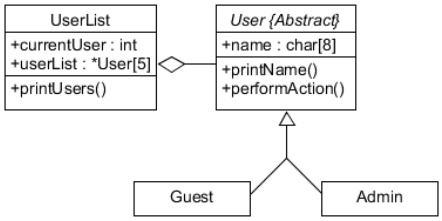

Figure 1: A UML class diagram of vlogin

2.1 Vulnerability

vlogin contains a contiguous memory corruption vulnerability when the user enters the username for the guest object. Usernames are stored in an array of chars of length 8. Taking into account the null terminator byte, this limits the length of usernames to 7 characters long. However, there are no bound checks on the size of user input, which means an attacker can overwrite an arbitrary-length section of memory proceeding the name field. This will be the entry point for injecting counterfeit objects. The following code shows where the memory corruption vulnerability is present in the source code.

```
char name[8] = {}; // In the User class
// Main function
Guest user;
std::cin >> temp.s;
memcpy(user.name, temp.s.data(), temp.s.length());
```

3 Attack Approach

The goal of this COOP attack is to inject a counterfeit Admin object into memory and perform an administrator-level action. The following steps summarize how the attack is carried out:

1. Reverse engineer vlogin to locate objects in memory, vtable pointers, and local variables.
2. Create a data-only payload that contains a counterfeit Admin object. The payload must also overwrite the currentUser variable, so it refers to the injected object.
3. Utilize the memory corruption vulnerability to inject the payload into memory.
4. Perform the administrator-level action.

3.1 Reverse Engineering

It's important to note that vlogin was compiled using the g++ compiler, in 32-bit mode, without ASLR, and with debugging symbols. While ASLR is certainly a reasonable defense mechanism, bypassing it using a memory disclosure vulnerability is outside the scope of this project, would detract from focusing on the COOP attack, and would make the reverse engineering and payload injection processes more complicated.

I began by debugging vlogin with GDB and reaching the state in the program just after when user input has been stored in the name field. Using the *info locals* and *print* commands, I was able to locate where the objects were stored in memory. This was an iterative process where I modified the source code to nudge objects into the desired location. For example, I created a wrapper class for a string variable in order to ensure that it was loaded into the correct place in memory. Appendix A shows a screenshot of GDB with the locations of objects in memory, including vtable locations, and a memory dump of the relevant section that will be overwritten with the payload. It's worth noting that the vtable for the Admin class is located at address 0x0804beb0. This will be referred to by the counterfeit object.

0xffffd13c	vtable for Guest
0xffffd140	User input name
0xffffd144	
0xffffd148	vtable for Admin
0xffffd14c	admin name
0xffffd150	
0xffffd154	vtable for Guest
0xffffd158	Alice name
0xffffd15c	
0xffffd160	vtable for Guest
0xffffd164	Bob name
0xffffd168	
0xffffd16c	vtable for Guest
0xffffd170	Charlie name
0xffffd174	
0xffffd178	currentUser variable
0xffffd17c	userList array
0xffffd180	
0xffffd184	
0xffffd188	
0xffffd18c	

Figure 2: Layout of objects in memory

From here, I constructed a payload that overwrites memory at addresses 0xffffd140 - 0xffffd178, that is, it overwrites User objects and the currentUser variable. Specifically, I replace the Guest object for Bob with an Admin object named w00t and set the currentUser variable to 2, which refers to this injected Admin object.

3.2 Results

In order to aid in the exploitation process, I created a payload injector program in C that writes an array of unsigned chars into stdout. Appendix C shows the COOP payload. It then gets piped into vlogin through a command in the command line. I also created a modified version of vlogin called vlogin-NoLoop which is identical to vlogin, except the main loop is replaced by calls to the functions that prints out the current users in the system and performs the action of the current user. This was done to simplify the payload injection process as the normal vlogin program would have required multiple user inputs to verify that the exploitation was successful. Appendix B shows a screenshot of a successful COOP attack on vlogin where an administrator-level action is performed.

4 Related and Future Work

The idea of Counterfeit Object-Oriented Programming was formalized by Schuster et al. [6] in their paper where they lay out the goals of COOP, a basic approach to creating a payload, and ways of preventing COOP. Ispoglou et al. [4] presented Block Oriented Programming (BOP), a framework for automating data-only attacks where payloads are expressed in a high-level language known as SPL. While meant as a tool for assessing the residual attack surface after implementing a CFI defense, BOPC is useful for stitching together chains of functional blocks using dispatcher blocks and determining where data needs to be injected into a program to execute a data-only payload. In terms of defense, Data Flow Integrity (DFI) [3] has been proposed but not yet seen any implementations due to the high overhead of tracking the flow of data in a program.

Future work could hardness the BOPC tool to run a simple SPL payload on top of vlogin. This extension could also showcase how to pass parameters through multiple virtual functions on overlaid counterfeit objects, a COOP feature not demonstrated in this attack. Overall, vlogin has the potential of being a learning tool for teaching the basics of COOP and BOP. As far as data-only attacks are concerned, future work could focus on detecting changes to data integrity within a program's memory space. This could be done with the use of hash functions or other detection mechanisms to ensure that counterfeit objects haven't been injected into memory. In order to mitigate COOP attacks that rely on tampered vtable pointers, a C++ compiler extension could be used to restrict the vtables that child classes are allowed to use.

5 Conclusion

In this paper I introduce vlogin, a vulnerable program meant to showcase how to carry out a data-only attack. By reverse engineering vlogin, I mapped out the locations of objects in memory and created a data-only payload that contained

a counterfeit Admin object which was used to perform an administrator-level action. While meant as a toy example, breaking this program requires understanding how C++ objects are stored in memory and thinking about which data fields may be useful for manipulation. vlogin serves as a prototype program for learning about COOP and data-only attacks.

6 Availability

The source code for vlogin, vloginNoLoop, and the injector program with the COOP payload, along with compiling instructions can be found on my website at: https://simongreenblatt.github.io/p/research/

7 Appendices

A Locations of Objects in Memory

Figure 3: The first object stored in memory is userList

B Successful COOP Attack

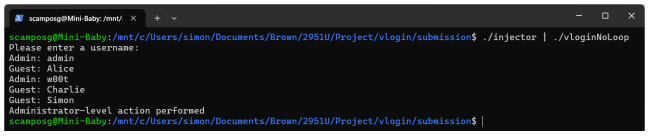

Figure 4: Successful exploitation with an injected counterfeit Admin object

C vlogin COOP Payload

```
unsigned char payload[] =
// Simon name
"\x53\x69\x6d\x6f\x6e\x00\x00\x00"
// vtable for Admin
"\xb0\xbe\x04\x08"
// admin name
"\x61\x64\x6d\x69\x6e\x00\x00\x00"
// vtable for Guest
"\xa0\xbe\x04\x08"
// Alice name
"\x41\x6c\x69\x63\x65\x00\x00\x00"
// vtable for Admin
"\xb0\xbe\x04\x08"
// w00t name
"\x77\x30\x30\x74\x00\x00\x00\x00"
// vtable for Guest
"\xa0\xbe\x04\x08"
// Charlie name
"\x43\x68\x61\x72\x6c\x69\x65\x00"
// currentUser variable set to w00t
"\x02\x00\x00\x00";
```

References

[1] Martin Abadi, Mihai Budiu, Ulfar Erlingsson, and Jau Ligatti. Control-flow integrity. In *12th ACM conference on Computer and communications security*, 2005. https://dl.acm.org/doi/10.1145/1102120.1102165.

[2] Nathan Burow, Xinping Zhang, and Mathias Payer. Sok: Shining light on shadow stacks. In *IEEE Symposium on Security and Privacy*, 2019. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8835389.

[3] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *7th symposium on Operating systems design and implementation.*, 2006. https://dl.acm.org/doi/10.5555/1298455.1298470.

[4] Kyriakos Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. *Block Oriented Programming, Automating Data-Only Attacks*. Association for Computing Machinery, 2018. https://dl.acm.org/doi/pdf/10.1145/3243734.3243739.

[5] Kangjie Lu, Chengyu Song, Byoungyoung Lee, Simon P. Chung, Taesoo Kim, and Wenke Lee. Aslr-guard: Stopping address space leakage for code reuse attacks. In *22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015. https://dl.acm.org/doi/10.1145/2810103.2813694.

[6] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit object-oriented programming. In *IEEE Symposium on Security and Privacy*, 2015. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7163058.