# zergRush : An Android Rootkit

Colton Kopsa

*<2018-02-15 Thu>*

Within the field of malware exists the desire to obtain root, originally known as *rootkits*. [1] With root access to a system, the user can do virtually anything they want in the system. [1] This malware has found use in the Android operating system, allowing users to gain more freedom with there devices at the cost of security. The *zergRush* malware was able to provide root access for all Android phones using version 2.2 (Frozen Yogurt) and 2.3 (Gingerbread) by means of the buffer overflow and use-after-free vulnerabilities. [2]

The buffer overflow vulnerability used by *zergRush* was found in the *FrameworkListener::dispatchCommand()* from the *libsysutils* library. [2]. This function allowed the user to input commands that the this function would dispatch for execution. [3] However, when copying a string from the user's entered command into a local c-string, the halting condition was based not on the size of the local c-string, rather it halted when a '0' was encountered, in other words it was expecting a 0-terminating c-string. [3] *zergRush* was able to exploit this by overflowing the buffer and using Return Oriented Programming to point to arbitrary code stored that was stored on the heap. [4, 5, 6]

## 1 ROP Gadgets

Modern operating systems implement countermeasures called data execution prevention (DEP) to prevent hackers from adding arbitrary code to a stack through user input. [7] It does this by marking a memory range (like what stores a user's input) as nonexecutable. [7] To thwart that protection, hackers use Return Oriented Programming. [7] "ROP takes its name from the RETN assembly instruction, the most important one for code executed via a stack-overflow attack because it's responsible for driving the process control out of the attacked process flow." [7] "For this property, each instruction sequence ending in the IA32 instruction “RETN” is called a *gadget.*" [7] By chaining these ROP gadgets together, its possible to execute virtually any program.

### 1.1 How it was used in *zergRush*

*zergRush* searches through the stack to find necessary ROP gadgets. [4] In *FrameworkListener::dispatchCommand()*, *zergRush* uses buffer overflow on the data array argument to write over the original return pointer with the start of the ROP gadget chain. [2, 3, 4] The ROP gadget chain sends execution of the program to the heap where the arbitrary malicious code is stored. [4] The running of this code provides the user root privileges. [4]

```
/*
 * add sp, #108 -> b01b
 * pop {r4, r5, r6, r7, pc} -> bdf0
 *
 * pop {r0, pc} -> bd01
 */
```

Figure 1: *find_rop_gadgets()* comment block from zergRush.c [4]

## 2 Use-after-free

The ability of *zergRush* to execute code on the heap was made possible by means of a use-after-free vulnerability. [5] This vulnerability can exist when memory on the heap is freed, but a pointer continues to reference the freed memory. "Referencing memory after it has been freed can cause a program to crash, use unexpected values, or execute code." [8] A typical example of this weakness can be as follows: "The memory in question is allocated to another pointer validly at some point after it has been freed. The original pointer to the freed memory is used again and points to somewhere within the new allocation. As the data is changed, it corrupts the validly used memory; this induces undefined behavior in the process." [8] In the event that a class is stored on the heap, its function pointers are also be st:ored on the heap. "If one of these function pointers is overwritten with an address to valid shellcode, execution of arbitrary code can be achieved." [8] In the case of *zergRush* it’s important to note that use-after-free can precede a buffer overflow. [8]

```
// Dynamic memory is allocated
char* ptr = (char*)malloc (SIZE);
if (err) {
  abrt = 1;
  // Dynamic memory is freed
  free(ptr);

 }
...
if (abrt) {
  // Dynamic memory is referenced after being freed
  logError("operation aborted before commit", ptr);

}
```

Figure 2: Example of *free-after-use* vulnerability
[8]

### 2.1 How it was used in *zergRush*

Within the source code for *zergRush* exists this comment: "Exploited by rewriting a *FrameworkCommand* object making the *runCommand* point to our first ROP gadget." [4] Android's *FrameworkCommands* are stored on the heap. [3] Because this data is stored on the heap, *zergRush* checks the registers r9, r10, and fp to determine if any of them contain a valid address to the heap. [4] When the *FrameworkCommands* are freed, *zergRush* keeps the pointer to its malicious *FrameworkCommand*, and when the ROP gadgets send the execution of the program to the heap it provides the user root privileges. [4, 3]

## 3 Response to *zergRush*

Because the entirety of *zergRush's* exploit was founded on their ability to overflow the buffer, Google's response was targeted at that vulnerability. [9] A bounds-check was added to the copying of the user's input to make sure that their input didn't cause an overflow. [9] If in the case that an overflow happened, the program would notify the user that they had exceeded the length of the buffer. [9] Though the use-after-free vulnerability remained, the ability to overwrite the return pointer to access the arbitrary code on the heap was removed making the exploit ineffective. [9]

*zergRush* was successful at exploiting Android. Its ability to overwrite the stack and modify the return pointer with ROP gadgets allowed it to gain control. Its exploitation of the use-after-free vulnerability and ability to write to the stack allowed it to insert arbitrary code. These exploits, when coupled, provided the user the ability to gain root privileges of a phone. However, these we're all made possible through a classic buffer overflow vulnerability.

## References

[1] James N. Helfrich. *Security for Software Engineers*, chapter 3 : Software Weapons - Rootkit, 6 : Memory Injection - Stack Smashing. 2018.

[2] Google. libsysutils rooting vulnerability ("zergrush"), 2011.

[3] Google. Frameworklistener.cpp, 2011.

[4] Revolutionary. revolutionary/zergrush.

[5] Common Vulnerabilities and Exposures. Cve-2011-3874.

[6] Nick Kralevich Dan J Rosenberg, Kurt Seifried. Cve request: Android: vold stack buffer overflow, 2011.

[7] Marco Prandini and Marco Ramilli. Return-oriented programming. *IEEE Security & Privacy*, 10(6):84–87, 2012.

[8] Common Weakness Enumeration. Cwe-416: Use after free, 2018.

[9] David 'Digit' Turner. Frameworklistener.cpp, 2011.