Matt Piotrowski

ECE 531

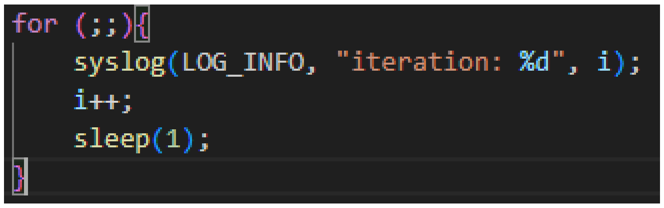
Professor Lamb

6/24/2023

Attack Surface Analysis

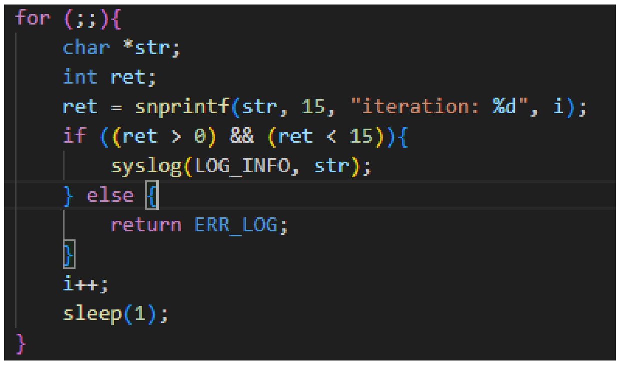
In this paper we will examine the attack surface of the daemon that we wrote in the last assignment and examine the ways in which we can harden our code against attack. The attack surface of a program describes the ways that an attacker can exploit our system. The exploits of the system usually occur on the input or output of the system but can also occur due to bugs in the program itself. Our daemon process begins at system startup and writes to the syslog every second with what iteration it is on and then stops at system shutdown.

The first attack surface that we will look at has to do with unsafe usage of syslog. All strings passed into syslog (user generated or not) should be passed in as a formatted string rather than just the direct buffer. An example of improper use is shown below in figure 1 and the correct use is shown in figure 2.



```
for (;;){
    syslog(LOG_INFO, "iteration: %d", i);
    i++;
    sleep(1);
}
```

Figure 1 Unsafe Code Example

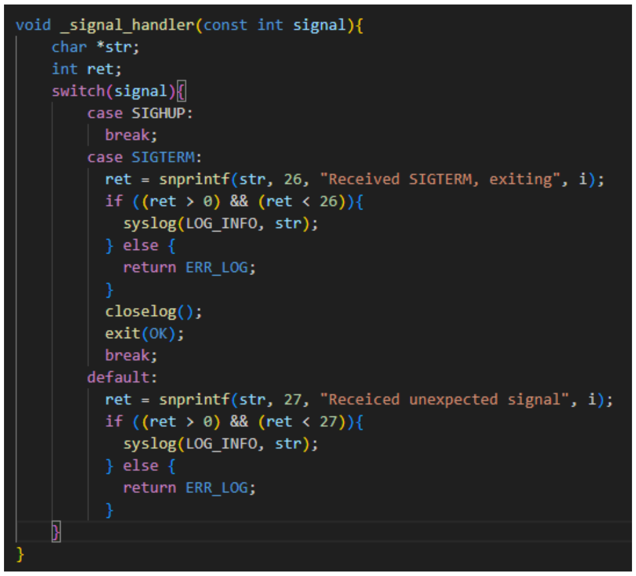


```
for (;;){
    char *str;
    int ret;
    ret = snprintf(str, 15, "iteration: %d", i);
    if ((ret > 0) && (ret < 15)){
        syslog(LOG_INFO, str);
    } else {
        return ERR_LOG;
    }
    i++;
    sleep(1);
}
```

Figure 2 Safe Code Example

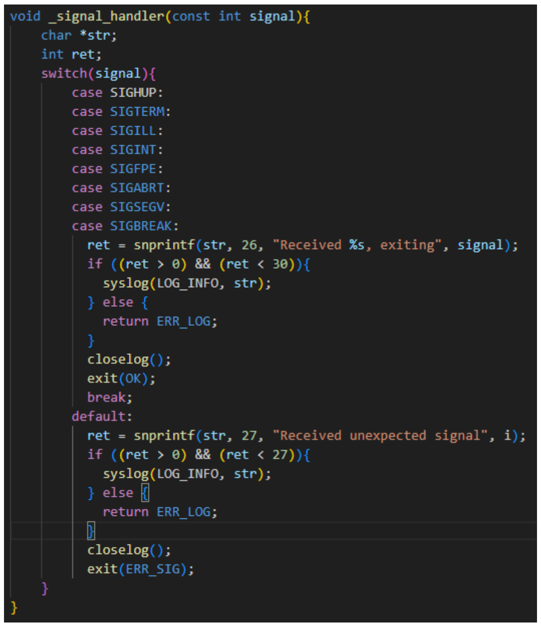
This safe string processing prevents attackers from using string injection attacks and buffer overflow attacks to exploit our system. CAPEC-67 further outlines the dangers of improper string handling with syslog (https://capec.mitre.org/data/definitions/67.html) .

The next attack surface we will examine is the signal processing of our daemon. Originally, we only explicitly handled a hangup signal and terminate signal. All other received signals would just print to syslog that an unexpected signal was received but not close anything off. This could be a problem if an attacker was able to send signals that should terminate the program but things like syslog are left open with root privileges enabled. Figure 3 below shows our original signal processing that does not handle all cases and figure 4 shows our enhanced processing that takes care of more signals.

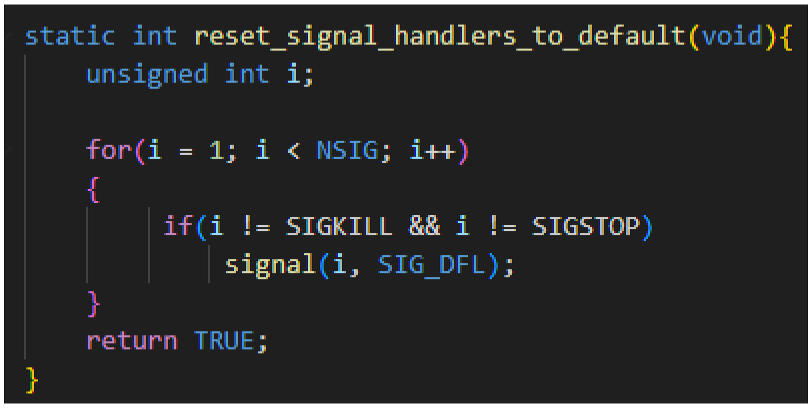


```c
void _signal_handler(const int signal){
    char *str;
    int ret;
    switch(signal){
        case SIGHUP:
          break;
        case SIGTERM:
          ret = snprintf(str, 26, "Received SIGTERM, exiting", i);
          if ((ret > 0) && (ret < 26)){
            syslog(LOG_INFO, str);
          } else {
            return ERR_LOG;
          }
          closelog();
          exit(OK);
          break;
        default:
          ret = snprintf(str, 27, "Receiced unexpected signal", i);
          if ((ret > 0) && (ret < 27)){
            syslog(LOG_INFO, str);
          } else {
            return ERR_LOG;
          }
    }
}
```

Figure 3 Original Signal Processing



```c
void _signal_handler(const int signal){
    char *str;
    int ret;
    switch(signal){
        case SIGHUP:
        case SIGTERM:
        case SIGILL:
        case SIGINT:
        case SIGFPE:
        case SIGABRT:
        case SIGSEGV:
        case SIGBREAK:
          ret = snprintf(str, 26, "Received %s, exiting", signal);
          if ((ret > 0) && (ret < 30)){
            syslog(LOG_INFO, str);
          } else {
            return ERR_LOG;
          }
          closelog();
          exit(OK);
          break;
        default:
          ret = snprintf(str, 27, "Received unexpected signal", i);
          if ((ret > 0) && (ret < 27)){
            syslog(LOG_INFO, str);
          } else {
            return ERR_LOG;
          }
          closelog();
          exit(ERR_SIG);
    }
}
```



```c
static int reset_signal_handlers_to_default(void){
    unsigned int i;

    for(i = 1; i < NSIG; i++)
    {
        if(i != SIGKILL && i != SIGSTOP)
            signal(i, SIG_DFL);
    }
    return TRUE;
}
```

Figure 4 Enhanced Signal Processing

As you can see, we added more signals to track as well as better handling of those signals to make sure that we close all necessary ports. We also added a function to return all signals back to their default so that we can ensure they were never changed. If they had been, an attacker could take advantage of that and the program may not execute as planned.

One of the easiest things we can do to harden our code is to include error processing for functions like fork() and chdir(). This ensures that we have expected behavior should a function behave differently than we expect.

It is impossible to defend against every attack surface that exists, as new exploits are being found daily, but that does not mean that it is not important to protect your system. It is extremely important to do the research on known exploits of libraries and functions your program uses and to make sure you push updates as needed.