Serial phishing a Mac OS X application for newbies

Target: MacDviX (http://www.kiffe.com/textools.html)
Tools used: OTX, GDB, 0xED
Platform: Mac OS X Leopard 10.5.6 @ Intel x86
Document version: 1.0 (23/02/2008)

Index:

0 - Introduction

Again by popular demand, let's learn how to phish a serial number (and keygen) from a simple application.
This target was suggested by an user and great target it is for our learning purposes !
It's serial algorithm is very easy and you can even patch it if you want (I will leave that as an exercise
so you can improve your skills!).

And now, let's start the fun !
fG!

1 - Attacking our target

Since you should already master gdb and otx from previous tutorial, we are going directly to our target.
As usual, the first approach is to understand the protection behaviour and what hints we might get so we can start working.

Load the application, and the first thing you get is a nag message box with a timer, telling you should input a password.
After the timer runs out, try to insert some password like our beloved 1234567890.
You will get a "Invalid entry." message. Fair enough, looks like a good hint.

Disassemble the main binary with OTX and search for that specific message, "Invalid entry". There is a single hit ! Hummm looks good!
Let me show you that piece of code (line numbers added for easy code reference):

```
(...)
1: 0000e837  e8441a0400                 calll       0x00050280
_strcpy
2: 0000e83c  891c24                     movl        %ebx,(%esp)
3: 0000e83f  e84ffaffff                 calll       _CheckPassword
4: 0000e844  84c0                       testb       %al,%al
5: 0000e846  754a                       jne         0x0000e892
6: 0000e848  c744240403000000           movl        $0x00000003,0x04(%esp)
7: 0000e850  893424                     movl        %esi,(%esp)
8: 0000e853  e864e4ffff                 calll       _SelectEditData
9: 0000e858  c744241038cd0200           movl        $0x0002cd38,0x10(%esp)
Invalid entry.
(...)
```

Even without starting to understand this piece of code, you should have already spotted an interesting call, the one to _CheckPassword.
So, we have a call to some function named CheckPassword, then we have a decision (testb,

jne combination). If the jump isn't executed, we find our bad serial message being moved into the stack.
We can try to assume that if that JNE is executed we might get a good serial message.
Assuming this, we might want to place our serial phishing bet into the CheckPassword function, because it should be the one responsable for serial number verification.

Let's find the disassembly for this function. Search the otx output for _CheckPassword.
There are four hits and you can easily find the code for it.
Here it is:

```
_CheckPassword:
0000e293  55                                          pushl          %ebp
0000e294  89e5                                        movl           %esp,%ebp
0000e296  57                                          pushl          %edi
0000e297  8b5508                                      movl           0x08(%ebp),%edx
0000e29a  85d2                                        testl          %edx,%edx
0000e29c  7427                                        je             0x0000e2c5
0000e29e  89d7                                        movl           %edx,%edi
0000e2a0  fc                                          cld
0000e2a1  b9ffffffff                        movl              $0xffffffff,%ecx
0000e2a6  b800000000                        movl              $0x00000000,%eax
0000e2ab  f2ae                                        repnz/scasb %al,(%edi)
0000e2ad  83f9f7                                      cmpl           $0xf7,%ecx
0000e2b0  7513                                        jne            0x0000e2c5
0000e2b2  807a022a                                    cmpb           $0x2a,0x02(%edx)
'*'
0000e2b6  750d                                        jne            0x0000e2c5
0000e2b8  807a0540                                    cmpb           $0x40,0x05(%edx)
'@'
0000e2bc  7507                                        jne            0x0000e2c5
0000e2be  b801000000                        movl              $0x00000001,%eax
0000e2c3  eb05                                        jmp            0x0000e2ca
0000e2c5  b800000000                        movl              $0x00000000,%eax
0000e2ca  5f                                          popl           %edi
0000e2cb  5d                                          popl           %ebp
0000e2cc  c3                                          ret
```

Load the application into gdb and set a breakpoint at 0xe293 (where it begins).
Start the application, wait for the timer and insert 1234567890 as password.
Our breakpoint should be enforced and gdb stops at 0xe293.
This is what we get:

```
Breakpoint 1, 0x0000e293 in CheckPassword ()
--------------------------------------------------------------------------[regs]
  EAX: 00000000  EBX: 0001685C  ECX: 00000004  EDX: 65747267  o d I t S z a P c
  ESI: 00045790  EDI: 000458D0  EBP: BFFFF598  ESP: BFFFF57C  EIP: 0000E293
  CS: 0017  DS: 001F  ES: 001F  FS: 0000  GS: 0037  SS: 001F
[001F:BFFFF57C]-------------------------------------------------------[stack]
BFFFF5CC : 00 00 00 00  00 00 00 00 - 00 00 00 00  00 00 00 00 ................
BFFFF5BC : 00 00 00 00  00 00 00 00 - 00 00 00 00  00 00 00 00 ................
BFFFF5AC : 00 00 00 00  00 00 00 00 - 00 00 00 00  00 00 00 00 ................
BFFFF59C : 97 23 01 00  05 00 00 00 - C0 16 03 00  00 00 00 00 .#..............
BFFFF58C : 5C 68 01 00  90 57 04 00 - D0 58 04 00  D8 F7 FF BF \h...W...X......
BFFFF57C : 76 E2 00 00  00 00 00 00 - 5C 68 01 00  2D 6C 6F 96 v.......\h..-lo.
[0017:0000E293]-------------------------------------------------------[code]
0xe293 <CheckPassword>: push   ebp
0xe294 <CheckPassword+1>:       mov    ebp,esp
0xe296 <CheckPassword+3>:       push   edi
0xe297 <CheckPassword+4>:       mov    edx,DWORD PTR [ebp+0x8]    <- interesting
0xe29a <CheckPassword+7>:       test   edx,edx                    <- interesting
0xe29c <CheckPassword+9>:       je     0xe2c5 <CheckPassword+50>  <- interesting
0xe29e <CheckPassword+11>:      mov    edi,edx
```

```
0xe2a0 <CheckPassword+13>:      cld
--------------------------------------------------------------------------------
```

This first interesting lines are 4, 5 and 6. Step until you reach address 0xe29a (line 5).

```
0x0000e29a in CheckPassword ()
--------------------------------------------------------------------------[regs]
  EAX: BFFFDAF0  EBX: BFFFDCF0  ECX: BFFFDD00  EDX: BFFFDCF0  o d I t s Z a P c
  ESI: 0019A400  EDI: 00000000  EBP: BFFFDAB8  ESP: BFFFDAB4  EIP: 0000E29A
  CS: 0017  DS: 001F  ES: 001F  FS: 0000  GS: 0037  SS: 001F
[001F:BFFFDAB4]-------------------------------------------------------[stack]
BFFFDB04 : 00 00 00 00  74 72 70 63 - 74 00 00 00  F0 B1 18 00 ....trpct.......
BFFFDAF4 : 35 36 37 38  39 30 00 00 - 8E BE 87 00  30 AC 19 00 567890......0...
BFFFDAE4 : 00 00 00 00  18 DB FF BF - 0A 00 00 00  31 32 33 34 ............1234
BFFFDAD4 : EC DA FF BF  48 DB FF BF - F0 B1 18 00  80 BE 87 00 ....H...........
BFFFDAC4 : F0 DC FF BF  74 78 65 74 - F4 01 00 00  F0 DC FF BF ....txet........
BFFFDAB4 : 00 00 00 00  08 DF FF BF - 44 E8 00 00  F0 DC FF BF ........D.......
[0017:0000E29A]-------------------------------------------------------[code]
0xe29a <CheckPassword+7>:       test   edx,edx
0xe29c <CheckPassword+9>:       je     0xe2c5 <CheckPassword+50>
0xe29e <CheckPassword+11>:      mov    edi,edx
0xe2a0 <CheckPassword+13>:      cld
0xe2a1 <CheckPassword+14>:      mov    ecx,0xffffffff
0xe2a6 <CheckPassword+19>:      mov    eax,0x0
0xe2ab <CheckPassword+24>:      repnz scas al,BYTE PTR es:[edi]
0xe2ad <CheckPassword+26>:      cmp    ecx,0xfffffff7
--------------------------------------------------------------------------------
```

Let's spy what's on EDX register. Try dumping that register as a string...

```
gdb$ x/s $edx
0xbfffdcf0:      "1234567890"
```

Voila, it's our serial number. Lines 4,5 and 6 are checking if input was empty or not. Recalling our disassembly:

```
_CheckPassword:
1:  0000e293  55                        pushl       %ebp
2:  0000e294  89e5                      movl        %esp,%ebp
3:  0000e296  57                        pushl       %edi
4:  0000e297  8b5508                    movl        0x08(%ebp),%edx <-
move our serial into EDX
5:  0000e29a  85d2                      testl       %edx,%edx      <-
check if EDX is empty
6:  0000e29c  7427                      je          0x0000e2c5     <-
jump if empty, else continue
7:  0000e29e  89d7                      movl        %edx,%edi
8:  0000e2a0  fc                        cld
9:  0000e2a1  b9ffffffff          movl        $0xffffffff,%ecx
10: 0000e2a6  b800000000          movl        $0x00000000,%eax
11: 0000e2ab  f2ae                      repnz/scasb %al,(%edi)
12: 0000e2ad  83f9f7                    cmpl        $0xf7,%ecx
13: 0000e2b0  7513                      jne         0x0000e2c5
14: 0000e2b2  807a022a                  cmpb        $0x2a,0x02(%edx)
'*'
15: 0000e2b6  750d                      jne         0x0000e2c5
16: 0000e2b8  807a0540                  cmpb        $0x40,0x05(%edx)
'@'
17: 0000e2bc  7507                      jne         0x0000e2c5
18: 0000e2be  b801000000          movl        $0x00000001,%eax
19: 0000e2c3  eb05                      jmp         0x0000e2ca
20: 0000e2c5  b800000000          movl        $0x00000000,%eax
21: 0000e2ca  5f                        popl        %edi
```

```
22: 0000e2cb  5d                                     popl         %ebp
23: 0000e2cc  c3                                     ret
```

Since our input isn't empty, you should continue to step until you have reached line 7, address 0xe29e.
At line 7, our serial is copied from register EDX to register EDI. No big deal here.
Line 8 clears the direction flag, causing string instructions to increment the SI and DI index registers.
Line 9 is filling ECX register with value 0xFFFFFFFF.
Line 10 is zeroing EAX register.
Line 11 is more interesting. Definition for repnz instruction is:
Repeats execution of string instructions while CX != 0 and the Zero Flag is clear.
CX is decremented and the Zero Flag tested after each string operation.
The combination of a repeat prefix and a segment override on processors other than the 386
may result in errors if an interrupt occurs before CX=0.

The scasb instruction definition is:
The x86 family of microprocessors come with with the scasb instruction which searches for the first occurence
of a byte whose value is equal to that of the AL register. The address of the start of the string itself has to
be in the EDI register. Technically, it is supposed to be in the extra segment, but we do not need to worry about
that in the flat 32-bit memory mode anymore. When used along with the repne prefix, the scasb instruction goes up
(or down, depending on the direction flag) the memory, looking for the match.

From these two definitions, you can understand that Line 11 is scanning for the NULL value. Why is that ?
EAX is a 32 bits register that can be divided into two 8 bits registers AH and AL.
The NULL value is equal to 0x00. Since EAX was zeroed on line 10, AH should give you 0x00 (an 8 bit value).
You can verify that in gdb:

```
gdb$ x/x (char) $al
0x0:    Cannot access memory at address 0x0
```

You need to typecast using (char) because $al is 8 bits or 1 byte. If you simply try x/x $al, you will get an error.

If you try to step the code, you will see that line 11 will keep being executed until it reaches the end of our serial
(strings end with the NULL value).
At line 12, ECX register is compared against 0xF7. You can recall that at line 7, ECX was filled with 0xFFFFFFFF.
Why is this ? Well from REPNZ definition you have:
"CX is decremented and the Zero Flag tested after each string operation."
So each time the repnz/scasb instruction was executed, the value in ECX was being decremented.

Now we finally can understand what these lines were doing, they were checking for the size of our serial number !!!
If you have used "1234567890" as serial number, check the value of ECX when you have reached line 12.

```
0x0000e2ad in CheckPassword ()
--------------------------------------------------------------------------[regs]
  EAX: 00000000  EBX: BFFFDCF0  ECX: FFFFFFF4  EDX: BFFFDCF0  o d I t s Z a P c
  ESI: 0019A400  EDI: BFFFDCFB  EBP: BFFFDAB8  ESP: BFFFDAB4  EIP: 0000E2AD
  CS: 0017  DS: 001F  ES: 001F  FS: 0000  GS: 0037  SS: 001F
[001F:BFFFDAB4]---------------------------------------------------------[stack]
BFFFDB04 : 00 00 00 00  74 72 70 63 - 74 00 00 00  F0 B1 18 00 ....trpct.......
```

```
BFFFDAF4 : 35 36 37 38  39 30 00 00 - 8E BE 87 00  30 AC 19 00 567890......0...
BFFFDAE4 : 00 00 00 00  18 DB FF BF - 0A 00 00 00  31 32 33 34 ............1234
BFFFDAD4 : EC DA FF BF  48 DB FF BF - F0 B1 18 00  80 BE 87 00 ....H...........
BFFFDAC4 : F0 DC FF BF  74 78 65 74 - F4 01 00 00  F0 DC FF BF ....txet........
BFFFDAB4 : 00 00 00 00  08 DF FF BF - 44 E8 00 00  F0 DC FF BF ........D.......
[0017:0000E2AD]-----------------------------------------------------------[code]
0xe2ad <CheckPassword+26>:      cmp    ecx,0xfffffff7
0xe2b0 <CheckPassword+29>:      jne    0xe2c5 <CheckPassword+50>
0xe2b2 <CheckPassword+31>:      cmp    BYTE PTR [edx+0x2],0x2a
0xe2b6 <CheckPassword+35>:      jne    0xe2c5 <CheckPassword+50>
0xe2b8 <CheckPassword+37>:      cmp    BYTE PTR [edx+0x5],0x40
0xe2bc <CheckPassword+41>:      jne    0xe2c5 <CheckPassword+50>
0xe2be <CheckPassword+43>:      mov    eax,0x1
0xe2c3 <CheckPassword+48>:      jmp    0xe2ca <CheckPassword+55>
--------------------------------------------------------------------------------
gdb$ x/x $ecx
0xfffffff4:     Cannot access memory at address 0xfffffff4
```

It's 0xFFFFFFF4, which of course will fail when compared against 0xFFFFFFF7.
I think you can spot that our serial number is 3 characters longer that what is expected.
Our test serial must be something like "123457". Try with this new one and check again the value of ECX.
This time you get:

```
gdb$ x/x $ecx
0xfffffff7:     Cannot access memory at address 0xfffffff7
```

The JNE at line 13 will be avoided and the first check is beaten.
Let's recall our disassembly:

```
_CheckPassword:
1:  0000e293  55                    pushl        %ebp
2:  0000e294  89e5                  movl         %esp,%ebp
3:  0000e296  57                    pushl        %edi
4:  0000e297  8b5508                movl         0x08(%ebp),%edx <-
move our serial into EDX
5:  0000e29a  85d2                  testl        %edx,%edx       <-
check if EDX is empty
6:  0000e29c  7427                  je           0x0000e2c5      <-
jump if empty, else continue
7:  0000e29e  89d7                  movl         %edx,%edi       <-
save our serial to EDI
8:  0000e2a0  fc                    cld                          <-
clear direction flag
9:  0000e2a1  b9ffffffff        movl         $0xffffffff,%ecx    <-
ECX = 0xFFFFFFFF
10: 0000e2a6  b800000000        movl         $0x00000000,%eax    <-
EAX = 0x00000000
11: 0000e2ab  f2ae                  repnz/scasb %al,(%edi)       <-
Scan for NULL value and at the same time calculting serial length
12: 0000e2ad  83f9f7                cmpl         $0xf7,%ecx      <-
Is input serial length equal to 7 chars ?
13: 0000e2b0  7513                  jne          0x0000e2c5      <-
Jump if not (invalid serial)
14: 0000e2b2  807a022a              cmpb         $0x2a,0x02(%edx)
'*'
15: 0000e2b6  750d                  jne          0x0000e2c5
16: 0000e2b8  807a0540              cmpb         $0x40,0x05(%edx)
'@'
17: 0000e2bc  7507                  jne          0x0000e2c5
18: 0000e2be  b801000000        movl         $0x00000001,%eax
19: 0000e2c3  eb05                  jmp          0x0000e2ca
```

```
20: 0000e2c5  b800000000                          movl            $0x00000000,%eax
21: 0000e2ca  5f                                        popl            %edi
22: 0000e2cb  5d                                        popl            %ebp
23: 0000e2cc  c3                                        ret
```

Let's look at line 14. The value 0x2a is being compared against some value at EDX. Remember that EDX still holds our serial. OTX shows us that 0x2a corresponds to ascii character *. It's easy to understand that some place in our serial is being compared against value 0x2a, or by other words, that place in our serial must hold the character *. The place in our serial should be character at position 3 (remember counting starts at 0 and not 1).
We can easily verify this:

```
gdb$
0x0000e2b2 in CheckPassword ()
--------------------------------------------------------------------------[regs]
  EAX: 00000000  EBX: BFFFD7A0  ECX: FFFFFFF7  EDX: BFFFD7A0  o d I t s Z a P c
  ESI: 0019A400  EDI: BFFFD7A8  EBP: BFFFD568  ESP: BFFFD564  EIP: 0000E2B2
  CS: 0017  DS: 001F  ES: 001F  FS: 0000  GS: 0037  SS: 001F
[001F:BFFFD564]---------------------------------------------------------[stack]
BFFFD5B4 : 30 A1 19 00  08 D7 FF BF - DA DB FF 91  D9 7A DB 93 0...........z..
BFFFD5A4 : 35 36 37 00  C8 D5 FF BF - 71 D3 FF 91  47 F3 1F 00 567.....q...G...
BFFFD594 : 00 D0 05 00  A8 D1 02 00 - 07 00 00 00  31 32 33 34 ............1234
BFFFD584 : 9C D5 FF BF  00 04 00 00 - 02 00 00 00  10 A1 19 00 ................
BFFFD574 : A0 D7 FF BF  74 78 65 74 - F4 01 00 00  A0 D7 FF BF ....txet........
BFFFD564 : 00 00 00 00  B8 D9 FF BF - 44 E8 00 00  A0 D7 FF BF ........D.......
[0017:0000E2B2]----------------------------------------------------------[code]
0xe2b2 <CheckPassword+31>:      cmp    BYTE PTR [edx+0x2],0x2a
0xe2b6 <CheckPassword+35>:      jne    0xe2c5 <CheckPassword+50>
0xe2b8 <CheckPassword+37>:      cmp    BYTE PTR [edx+0x5],0x40
0xe2bc <CheckPassword+41>:      jne    0xe2c5 <CheckPassword+50>
0xe2be <CheckPassword+43>:      mov    eax,0x1
0xe2c3 <CheckPassword+48>:      jmp    0xe2ca <CheckPassword+55>
0xe2c5 <CheckPassword+50>:      mov    eax,0x0
0xe2ca <CheckPassword+55>:      pop    edi
--------------------------------------------------------------------------------
gdb$ x/s $edx+0x2
0xbfffd7a2:      "34567"
gdb$ x/c $edx+0x2
0xbfffd7a2:     0x33
```

Conclusion: Our serial character number 3 should be equal to *.

At line 16, you have similar code. This time character is @ and our serial position is number 6.
This time input your serial like "12*4567" and check.

```
0x0000e2b8 in CheckPassword ()
--------------------------------------------------------------------------[regs]
  EAX: 00000000  EBX: BFFFD7A0  ECX: FFFFFFF7  EDX: BFFFD7A0  o d I t s Z a P c
  ESI: 0019A400  EDI: BFFFD7A8  EBP: BFFFD568  ESP: BFFFD564  EIP: 0000E2B8
  CS: 0017  DS: 001F  ES: 001F  FS: 0000  GS: 0037  SS: 001F
[001F:BFFFD564]---------------------------------------------------------[stack]
BFFFD5B4 : 50 6F 17 00  08 D7 FF BF - DA DB FF 91  E2 8E DB 93 Po..............
BFFFD5A4 : 35 36 37 00  C8 D5 FF BF - 71 D3 FF 91  E3 EE 1F 00 567.....q.......
BFFFD594 : 00 D0 05 00  A8 D1 02 00 - 07 00 00 00  31 32 2A 34 ............12*4
BFFFD584 : 9C D5 FF BF  00 04 00 00 - 02 00 00 00  30 6F 17 00 ............0o..
BFFFD574 : A0 D7 FF BF  74 78 65 74 - F4 01 00 00  A0 D7 FF BF ....txet........
BFFFD564 : 00 00 00 00  B8 D9 FF BF - 44 E8 00 00  A0 D7 FF BF ........D.......
[0017:0000E2B8]----------------------------------------------------------[code]
0xe2b8 <CheckPassword+37>:      cmp    BYTE PTR [edx+0x5],0x40
0xe2bc <CheckPassword+41>:      jne    0xe2c5 <CheckPassword+50>
0xe2be <CheckPassword+43>:      mov    eax,0x1
```

```
0xe2c3 <CheckPassword+48>:      jmp    0xe2ca <CheckPassword+55>
0xe2c5 <CheckPassword+50>:      mov    eax,0x0
0xe2ca <CheckPassword+55>:      pop    edi
0xe2cb <CheckPassword+56>:      pop    ebp
0xe2cc <CheckPassword+57>:      ret
--------------------------------------------------------------------------------
gdb$ x/s $edx+5
0xbfffd7a5:      "67"
gdb$ x/c $edx+5
0xbfffd7a5:     0x36
gdb$
```

After this check, 0x1 is moved into EAX and we return from CheckPassword function.
Moving 0x1 into EAX is usually a sign of a good serial.
Since no other checks are done, we can conclude that serial must be 7 chars long, 3rd char must be equal to * and 6th char must be equal to @.
All the other chars can be whatever we want.
Try to input the following serial, 12*45@7 or ab*de@g or any other combination. It should work :)

Final recall to disassembly listing:

```
_CheckPassword:
1:  0000e293  55                                 pushl          %ebp
2:  0000e294  89e5                               movl           %esp,%ebp
3:  0000e296  57                                 pushl          %edi
4:  0000e297  8b5508                             movl           0x08(%ebp),%edx <-
move our serial into EDX
5:  0000e29a  85d2                               testl          %edx,%edx       <-
check if EDX is empty
6:  0000e29c  7427                               je             0x0000e2c5      <-
jump if empty, else continue
7:  0000e29e  89d7                               movl           %edx,%edi       <-
save our serial to EDI
8:  0000e2a0  fc                                 cld                            <-
clear direction flag
9:  0000e2a1  b9ffffffff                 movl          $0xffffffff,%ecx        <-
ECX = 0xFFFFFFFF
10: 0000e2a6  b800000000                 movl          $0x00000000,%eax        <-
EAX = 0x00000000
11: 0000e2ab  f2ae                               repnz/scasb %al,(%edi)         <-
Scan for NULL value and at the same time calculting serial length
12: 0000e2ad  83f9f7                             cmpl           $0xf7,%ecx      <-
Is input serial length equal to 7 chars ?
13: 0000e2b0  7513                               jne            0x0000e2c5      <-
Jump if not (invalid serial)
14: 0000e2b2  807a022a                           cmpb           $0x2a,0x02(%edx)
<- compare our serial character number 3 against character *
15: 0000e2b6  750d                               jne            0x0000e2c5      <-
if not equal then jump (invalid serial)
16: 0000e2b8  807a0540                           cmpb           $0x40,0x05(%edx)
<- compare our serial character number 6 against character @
17: 0000e2bc  7507                               jne            0x0000e2c5      <-
if not equal then jump (invalid serial)
18: 0000e2be  b801000000                 movl          $0x00000001,%eax        <-
return our serial as a good one
19: 0000e2c3  eb05                               jmp            0x0000e2ca      <-
return to the end of the function
20: 0000e2c5  b800000000                 movl          $0x00000000,%eax        <-
return our serial as a bad one
21: 0000e2ca  5f                                 popl           %edi
22: 0000e2cb  5d                                 popl           %ebp
```

```
23: 0000e2cc  c3                                      ret
```

A keygen can be created easily. You just need some random digits for all the other positions. I leave that as an exercise for you.

2 - Conclusion

Here we are at the end. This tutorial teached you (hopefully!) how to fish a valid serial number. The biggest difficulty while phishing serials is to understand what the code is doing. You will need to have assembler knowledge and be able to understand what the code is doing, that is, reversing, transforming low level code (assembler) into high level (C or any other language, or at least an algorithm).

As an exercise, you could patch the program to accept any serial or to remove that initial nag. Try to do it :)

If you have any suggestions, doubts or found any error, please feel free to leave a comment at my blog http://reverse.put.as or drop an email at reverse AT put.as

Have fun!
fG!