Control Flow Integrity with *ptrace()*

Sebastian Krahmer *krahmer@suse.de*

May 18, 2006

Abstract

Control Flow Integrity (CFI) is a new technology to ensure a certain execution flow of a program. One of its aims is to prevent attacks against programs such as buffer overflow attacks. A sample implementation for Linux using the *ptrace()* mechanism is explained in this paper.
Search-engine-tag: krahmer-bbpaint-2006

1 Introduction

The structure of a program can be described in various ways. Statically, the source code describes exactly what it will do for certain input at runtime. Control flow graphs which are generated from the source code are just a different view of the same structure. At runtime, the binary code defines what the program, now layed out in memory, will do. Even at this time, it is possible to generate graphs in different ways to describe certain flows or to make certain structures of the flow visible. One such way has been described in [4]. Another (different) way is to record the execution of so called *basic blocks* of binary code. These blocks consist of opcodes at contiguous addresses, for example a sequence of instructions at assembler level without a branch or any other disruption of the instruction pointer. Since programs are deterministic, the number, position in memory and order of execution are fixed for the same input[1]. This results in a unique structure for every program which allows for the detection of the execution of code which has not been there before, for example code execution due to a buffer overflow exploit or a computer virus.
The nodes of the flow-graph are the basic blocks and contain the disassembled code executed within this block. The edges are the branch instructions between these blocks. Loops can easily be detected by self-reference. Multiple edges between the same blocks in the same direction can be collapsed [2] to shrink the size of the output file and to make the drawing significantly faster.

[1] Note that signals are also input.
[2] With help of the *dot2dot.pl* script which is part of the implementation available at [5].

Figure 1: A basic block from the libc vsyscall page.

```
0xffffe410: pop    ebp
0xffffe411: pop    edx
0xffffe412: pop    ecx
0xffffe413: ret
```

Figure 2: A basic block from the *libc*

```
0x400d7510: cmp    dword [0x0000000c], (8)0x00000000
0x400d7518: j(ne,nz)    byte (8)0x00000022
0x400d751a: push    ebx
0x400d751b: mov    edx, dword [esp + (8)0x00000010]
0x400d751f: mov    ecx, dword [esp + (8)0x0000000c]
0x400d7523: mov    ebx, dword [esp + (8)0x00000008]
0x400d7527: mov    eax, (32)0x00000004
0x400d752c: call dword [0x00000010]
```

2 The basic block structure

The basic block graph from a program can be obtained statically in two ways: by analyzing the binary image[3] or by tracing the control flow of the program at runtime. The resulting graphs may differ for the following reasons. The program used for tracing and recording the basic blocks structure [5] detects control flow changes (for example branches) by comparing the address calculated for the next instruction with the current address. This creates separate basic blocks such as in Figure 1, because, when the execution flow returns from a syscall, the address calculated for the next instruction does not match the instruction pointer.

With static analysis this block will not appear. On the other hand, conditional branches may be detected which could be used theoretically, but are indeed never executed as the condition will never or very seldom be true. An example is given in Figure 2 where a `jne` is never executed.

From [15] it is known that the question of whether a branch will ever be executed is undecidable in general.

The difference between the static and the dynamic basic block structures is not really important as the running program is deterministic. Even small changes in the graph, for example additional nodes and edges can be accepted if their number is below a certain threshold.

[3]This includes disassembling.

3 Detecting control flow changes

For the detection of changes to a program a certain threshold is set. The exact value of this threshold is determined empirically. It will influence the sensitivity of the detection. To detect changes which go beyond this threshold, the structure of the running program has to be recorded in a first step. This is as simple as attaching the tracer program [5] to the process which should be protected. Attaching the tracer to the *test* program from Figure 3 yields an output-file in the dot-format which can be converted to the graph shown in Figure 4. Whenever the *test* program is run, it will produce exactly this graph, except if someone subverts the behavior, for example using a buffer overflow attack. In this case the graph would look quite different depending on the new code blocks which are executed. Figures 6 and 7 which have been produced by the same program can serve as an instructive example.
Once the structure of the program has been saved, it can be loaded in a second step, whenever the tracer is attached again to the *test* process. It will then compare whether or not new basic blocks appear. Appropriate action, such as shutting down the process, can then be taken.
The *test* process has a very simple structure. It does not contain buffer overflow conditions. Examples for detected attacks are shown below.
Real-world programs are of far greater complexity as can be seen in their graphs, such as the one from the *top* process in Figure 5. For larger programs the probability grows that there are basic blocks which are not executed because a certain functionality is never or only rarely requested, such as particular menu items, configure options or protocol add-ons. If they are not seen during the recording phase, the tracer program will accidently see them as new blocks in the protection phase. This is not a problem if the threshold is set correctly, for example, it may be acceptable that there appear 10% [4] new basic blocks during a run; on the other hand, if new blocks occur in large numbers and with about the same complexity as the program just traced, then it is likely something is wrong. Nevertheless, while recording the structure, as much functionality as possible should be seen so that the resulting graph is as complete as possible.

4 Usage details

The program to record and match the basic block graphs can be found at [5]. Graphs are saved in the graphivz [6] dot-file format but can be converted to the VCG format with the *dot2vcg.pl* perl script [5]. The program is proof of concept. It is slow but has already been improved to use caching mechanisms to avoid multiple analysis of the same blocks. The block-matching in the protection phase is very basic. The addresses of the basic blocks are loaded and whenever a new

[4] It is obvious that the number of never executed blocks is small compared to the blocks executed during a run, if all functionality of the program is requested. However to have a strong value for a threshold further statistical research and experiments are necessary.

Figure 3: The source code of the *test* program.

```
 1    #include <stdio.h>


 2    int f()
 3    {
 4        write(2, "hello world\n", 13);
 5        return g();
 6    }

 7    int g()
 8    {
 9        usleep(1000);
10        return 2;
11    }

12    int main()
13    {
14        char buf[1024];

15        printf("%d\n", getpid());
16        sleep(10);
17        for (;;) {
18            printf("hello world\n");
19            sleep(3);
20            g();
21            f();
22            g();
23        }
24        return 0;

25    }
```

block is entered it is checked whether this block is known. A threshold could be added very easily. Even far more complex scenarios could be added such as sub-graph detection or the weight of nodes. This will, however, slow down the process. The detection of new basic blocks is a reliable and fast way to detect execution of new code.

5 Buffer Overflow example

The following program, which contains a buffer overflow condition, will be used as an example of how the basic block structure is recorded during a normal program run and how an attack can be detected.

The program is a simple server which accepts connections via the TCP/IP protocol. Once a connection arrived, a new process is spawned to handle this connection. The basic block structure of this program will be recorded while it is handling the connection of a client. The job of the server is to read data from the network and to answer with an *OK*. While reading the data from the network, a buffer overflow can be triggered. This allows attackers to construct evil clients which execute commands via this server. Since one can record the normal execution flow of the server as shown below, it is possible to detect the attack during a second run by comparing the two execution flows which should be identical if no attack happened, but which are quite different.

```
 1  #include <stdio.h>
 2  #include <netinet/in.h>
 3  #include <sys/socket.h>
 4  #include <sys/types.h>
 5  #include <errno.h>
 6  #include <unistd.h>
 7  #include <arpa/inet.h>
 8  #include <stdlib.h>
 9  #include <string.h>
10  #include <sys/wait.h>
11  #include <sys/mman.h>


12  void die(const char *s)
13  {
14      perror(s);
15      exit(errno);
16  }



17  int handle_connection(int fd)
18  {
19      char buf[1024];

20      write(fd, "OF Server 1.0\n", 14);
21      read(fd, buf, 4*sizeof(buf));
22      write(fd, "OK\n", 3);
23      return 0;
24  }


25  void sigchld(int x)
26  {
27      while (waitpid(-1, NULL, WNOHANG) != -1);
28  }


29  int main()
30  {
31      int sock = -1, afd = -1;
32      struct sockaddr_in sin;
33      int one = 1;

34      printf("&sock = %p system=%p mmap=%p\n", &sock, system, mmap);
```

```
35        if ((sock = socket(PF_INET, SOCK_STREAM, 0)) < 0)
36            die("socket");
37        memset(&sin, 0, sizeof(sin));
38        sin.sin_family = AF_INET;
39        sin.sin_port = htons(1234);
40        sin.sin_addr.s_addr = INADDR_ANY;

41        setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &one, sizeof(one));

42        if (bind(sock, (struct sockaddr *)&sin, sizeof(sin)) < 0)
43            die("bind");
44        if (listen(sock, 10) < 0)
45            die("listen");

46        signal(SIGCHLD, sigchld);

47        for (;;) {
48            if ((afd = accept(sock, NULL, 0)) < 0 && errno != EINTR)
49                die("accept");
50            if (afd < 0)
51                continue;
52            if (fork() == 0) {
53                close(sock);
54                handle_connection(afd);
55                exit(0);
56            }
57            close(afd);
58        }

59        return 0;
60    }
```

Once the server process has been started, the tracer program has to be attached to record the basic block structure:

```
linux:bbpaint # ./server
&sock = 0xbffff37c system=0x80485e4 mmap=0x80485b4

[1]+  Stopped                 ./server
linux:bbpaint # bg
[1]+ ./server &
linux:bbpaint# ps aux|grep server
root   7111  0.0  0.1  1308  308 pts/8    S    09:20   0:00 ./server
root   7113  0.0  0.3  2676  692 pts/8    S+   09:21   0:00 grep server
linux:bbpaint # ./bbpaint -d server.dot -p 7111
New block: 7111 0
New block: 7111 ffffe403
New block: 7111 ffffe410
New block: 7111 400e6021
New block: 7111 80488d3
New block: 7111 80488b0
New block: 7111 8048594
New block: 7111 8048554
New block: 7111 4000c9e0
New block: 7111 4000cb60
New block: 7111 40012bab
New block: 7111 4000cb76
New block: 7111 40007e40
New block: 7111 40007e53
New block: 7111 40007a40
New block: 7111 40007aab
New block: 7111 40007e67
New block: 7111 40007af0
New block: 7111 40007b12
...
New block: 7111 400e6000
New block: 7129 400b4c41
New block: 7129 400b4c48
New block: 7129 4008bed0
New block: 7129 4008bed8
New block: 7129 400b4c74
New block: 7129 400b4ca7
New block: 7129 4008bef0
New block: 7129 400b4cae
New block: 7129 400b4c76
New block: 7129 4008bf10
...
```

This shows how the tracing process is learning the addresses and their relations while a legitimate client is connecting to the server. The tracer program

prints out the process ID of the code from which it is currently handling a block and the address of that block. As the server is handling new connections in a different process, a new process ID appears during the run. The output of this run eventually results in the graph of Figure 6.
Once one has learned all possible blocks, new clients will not trigger the execution of new or different blocks or execution in a different order, as long as they do not try to exploit the overflow condition. The tracer process can then be attached to the server process again. This time, the recorded basic block structure from the last run is loaded to match it against the executed blocks:

```
linux: bbpaint # ./bbpaint -D server.dot -p 7111
Loaded 207 basic blocks.
```

There will be only few or no new basic blocks if new clients connect and the server process is doing its normal job. There can be a few new basic blocks due to signal handling which interrupt the normal code flow. The amount, however, should be less than the threshold. The number of basic blocks for a normal program is very large compared to the number of new blocks during a normal program run.
However there is a risk if detection of attacks is only based on the occurrence of new blocks since attackers could potentially re-use existing code blocks. Strong CFI implementations should use additional metrics, such as the execution path of blocks or the weight of the nodes, to fend off attacks which try to stay below the new-blocks threshold by slowly making the tracer learning new blocks.
Figure 6 shows the basic blocks of the *server* process during a normal run while the graph in Figure 7 shows the same process while being successfully attacked with an exploit executing a shell. The difference between the two is obvious. If everything were OK then there would not be a difference.

6 Tracing child processes

As seen in the output of the tracer-program above, the tracing program which protects the server process has to take care of spawned child processes too. Otherwise it will not see the basic blocks which actually handle the input nor the different execution flows triggered by attackers.

Programs with CFI protection need to be attached to by the tracer. Most of these programs will handle untrusted input such as networking daemons. These daemons usually *fork()* off a child for each new connection. Newer kernels allow one to set the PTRACE_O_TRACEFORK and PTRACE_O_TRACEEXEC options in order to follow *fork()* and *execve()* system calls. This is necessary since the child process of the networking daemon also needs the protection as it actually handles the input data.
By this requirement the tracing process must distinguish between different registers (the instruction pointer in particular) for the processes. The list of basic blocks already handled and used for caching and also the disassembled code can

be shared by the processes as *fork()* implements copy on write and the `.text` segment is not written to.
The *execve()* system call however is much more complicated. On the one hand it is necessary to follow it to detect the creation of new blocks if a shell is executed for example. On the other hand it is not possible to distinguish between normal operations and attacks if the process, which is being traced, executed CGI or shell-scripts and so on all the time. This will overlay the graph and this is significant for the attack. It is not possible to protect processes which execute other complex programs, e.g. interpreters reliably yet.

7 Control Flow Integrity and Virus protection

CFI can be used to detect attacks against programs such as buffer overflows, computer viruses and other kinds of attack. Viruses usually infect the binary image of programs and add their own code in such a way that it is executed upon start of the infected program. Computer viruses also consist of basic blocks which need to be executed in order for the virus to work. There are two possible ways to use CFI for virus protection:

- Signature detection
  This is probably the weaker variant. Viruses are usually small pieces of code but also consist of a particular basic block structure which can be seen as a signature which can be detected upon execution. There are two problems with this approach: viruses might use techniques such as polymorphism which change the structure of the basic blocks and may make detection of signatures impossible. Additionally the addresses of the blocks probably differ on each infection since viruses usually contain position independent code. This can be resolved by using other metrics than addresses of the blocks in the same way as CFI would work for a randomized address space. Self encryption by the virus would not protect it from detection by the CFI engine since the basic block structure appears during execution of the decrypted code.
- White listing
  This is basically the same way as the one by which buffer overflows would be detected; one has to keep in mind that the attack is performed locally. For every program a description file can be placed in a special directory which must be protected by the Operating System from modification. The description file contains the basic block graph and has to be complete in order to avoid false positives. A virus which infects the program changes the flow-graph. Upon execution of the program the Operating System has to ensure that code can only be executed as permitted by the original flow-graph. Nodes added by the virus will be detected regardless of whether the virus is self-encrypting or polymorphic or of any other kind. The virus should, of course, not be able to modify the description file. This can be achieved by putting it onto a read-only file system.

8 Drawbacks and Problems

This paper developed a way to implement Control Flow Integrity on the Linux operating system. It could be implemented on any Operating System. One only needs to adapt the debugging mechanism to properly attach to the processes which need to be protected. The main drawback of this technique is, of course, the slowdown of the process since it runs in debugging mode. In the future it might be possible to add support by the kernel or even CPU support for debugging traps at branch instructions, to avoid the reliance on single stepping. Newer CPUs support the so-called *Branch Trace Flag* (BTF) in the debugging register which will cause debugging traps only on branches [11]. It is also possible to perform some binary translation during program loading to obtain the addresses of the branch instructions. Intermediate Language (IL) and Virtual Machine (VM) technology, as used in Java and C#, add another execution layer which can be used to create and match control flow graphs.

In principle: for CFI to work properly one needs (better) support from the kernel (speedup, child tracing, traps etc) and possibly even from the CPU. For VM and IL systems this is not needed.

The detection of new basic blocks does not work if the address space is randomized [5] since the blocks will have different addresses at each startup, so the structure saved from the previous run is useless as long as the detection is done by comparing the addresses of the basic blocks. However it might be possible to match the graphs or to recognize new blocks nevertheless using metrics, different from that comparing the addresses of the basic blocks,such as their length, crc32 or a hash of the blocks.

For a working CFI system based on *ptrace()* it is necessary that the attached process runs with privileges different from those of the tracing process. Otherwise the attacker could kill the protection process easily during the attack. If the protected process needs to run as root one needs special support from the kernel such as Role Based Access Control to defend against this attack.

9 Related Work

There is already work done by others in the field of CFI and program protection [9], [13]. Some also deal with basic blocks, in particular [12] which defines stronger goals in order to prevent attacks and which focuses on buffer overflows. Their implementation is not based on *ptrace()*. Control Flow Integrity can also be performed on the syscall level as shown in [10].

[5] Certain kernel-patches randomize addresses of the mapped code in order to make exploitation of buffer overfbws harder.

10 Conclusion

In this paper a method to implement CFI on a *ptrace()* basis has been described. Using the debugging hook from the kernel, it is possible to record and match the basic block structure of a process. This will work as long as the addresses are fixed, e.g. no address space randomization is used for the `.text` segments. For a randomized address space, other metrics can be used to detect attacks. The speed of the recording- and detection-phase can be increased dramatically if certain CPU and Operating System support is added such as single stepping on branches. This will allow one to build a very strong mechanism for preventing attacks against programs such as viruses or buffer overflow attacks. Sample graphs of traces are shown in Figures 4 and 5.

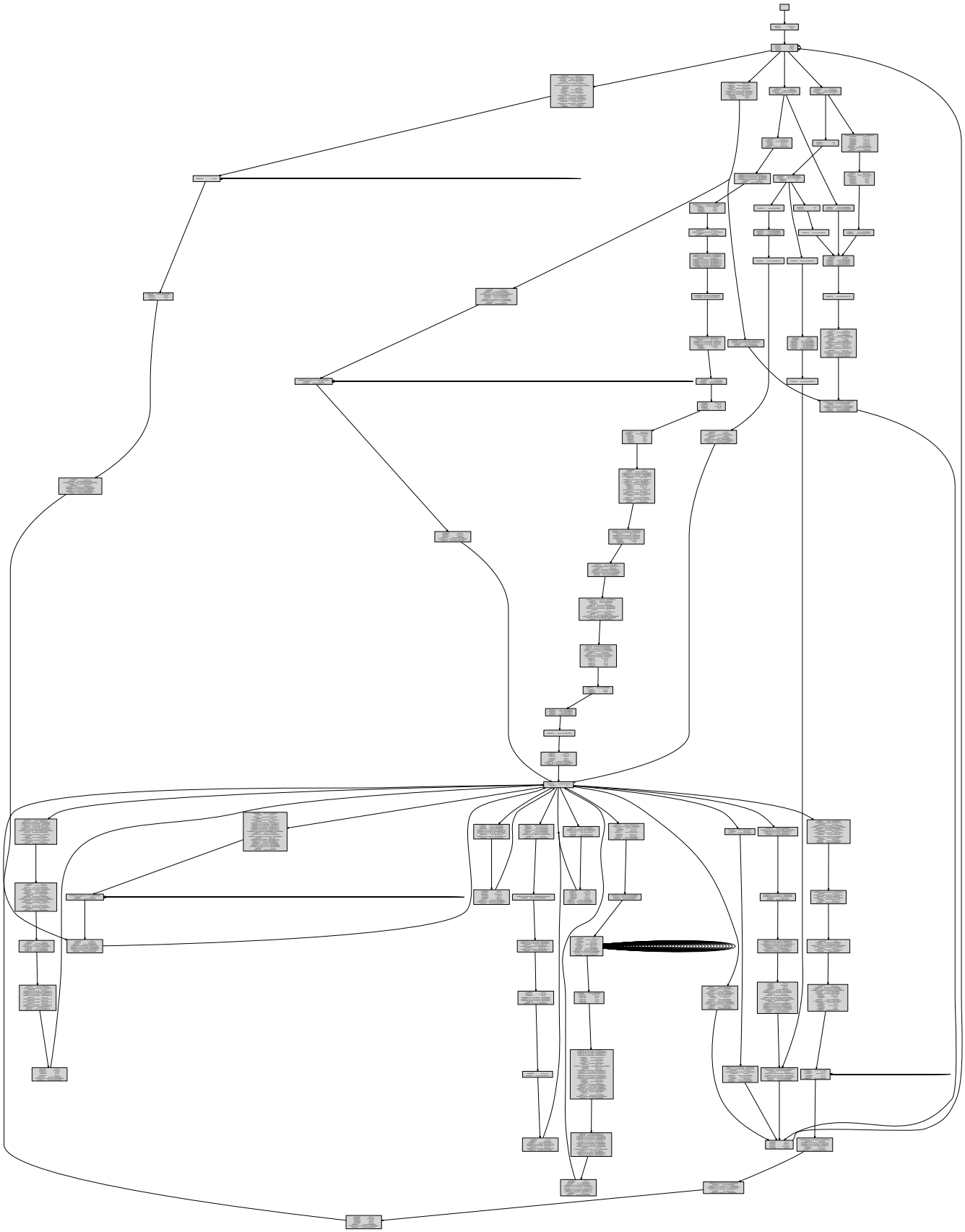

Figure 4: Basic blocks from the *test* process.

Figure 5: Basic blocks from the *top* process.

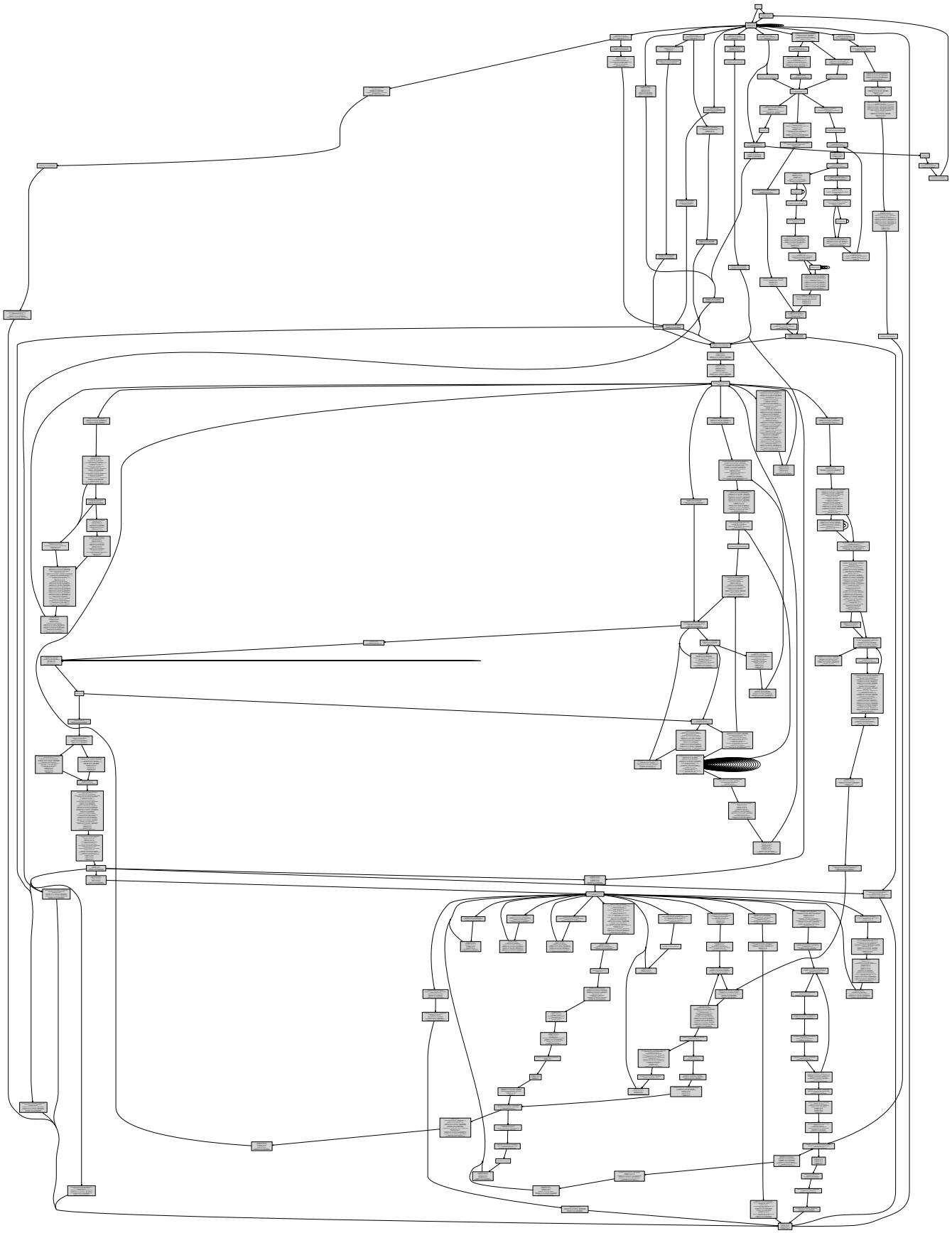

Figure 6: Basic blocks from the example *server* process if running regular without attacks.

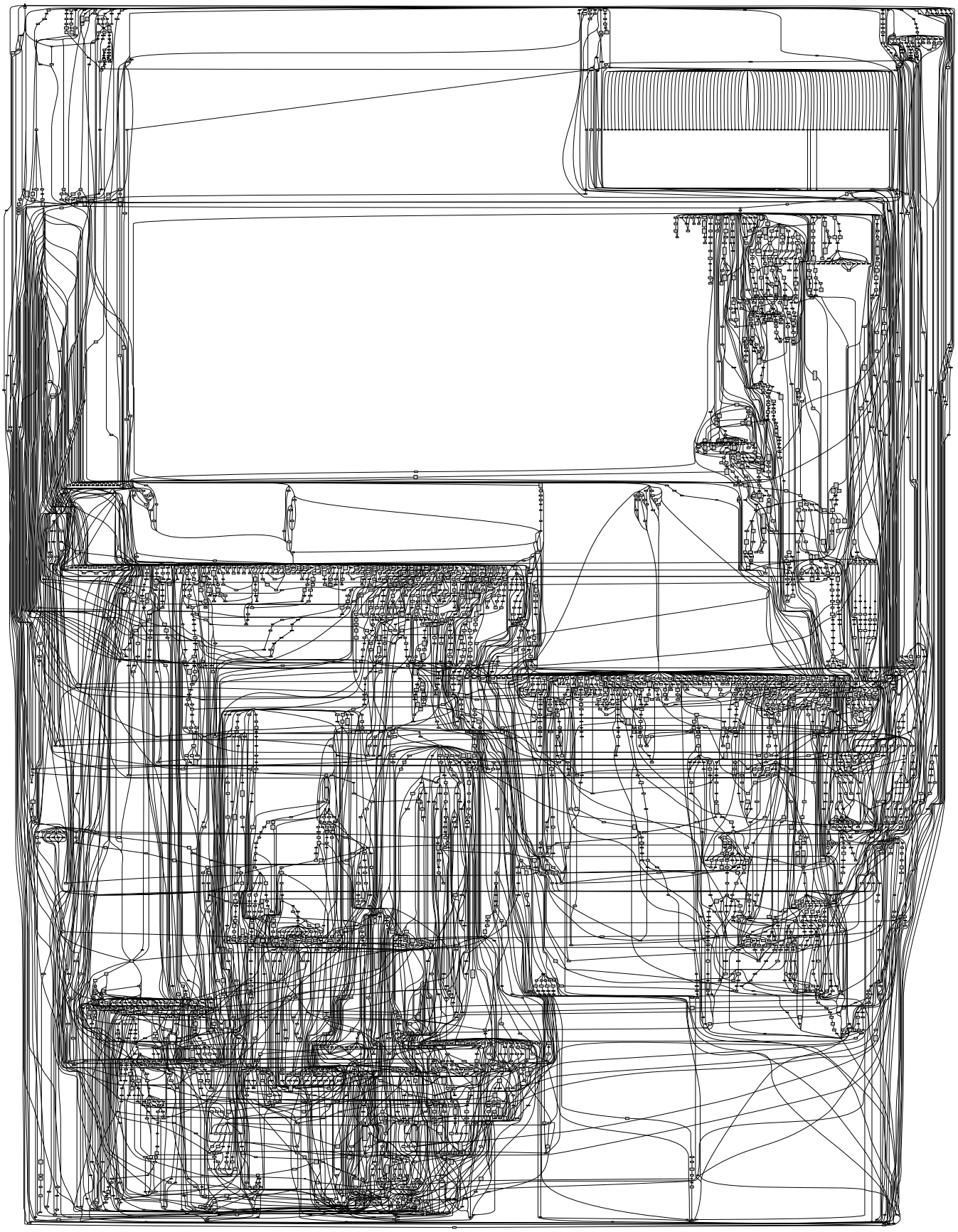

Figure 7: Basic blocks from the example *server* process if attacked with an exploit. Output of the tracing process has been filtered through *dot2dot.pl*.

References

[1] GCC:

`http://gcc.gnu.org`

[2] Sabre-security:

`http://sabre-security.com`

[3] aisee:

`http://aisee.com`

[4] instrumental:

`http://www.suse.de/~krahmer/instrumental`

[5] bbpaint:

`http://www.suse.de/~krahmer/bbpaint`

[6] graphviz:

`http://www.research.att.com/sw/tools/graphviz`

[7] Perl:

`http://www.perl.com/`

[8] VCG:

`http://rw4.cs.uni-sb.de/users/sander/html/gsvcg1.html`

[9] Martin Abadi, Mihai Buido, Ulfar Erlingsson, Jay Ligatti: *Control-Flow Integrity*, 2005

[10] Systrace:

`http://www.citi.umich.edu/u/provos/systrace/`

[11] IA-32 docs: *IA-32 Intel Architecture Software Developers Manual VOLUME 3*

[12] Vladimir Kiriansky, Derek Bruenning, Saman Amarasinghe: *Secure Execution Via Program Shepherding*, In *Proceedings of the 11th USENIX Security Symposium San Francisco 2002*

[13] J.Bergeron, M. Debbabi, M.M.Erhioui, B. Ktari: *Static Analysis of Binary Code to Isolate Malicious Behaviors*, 6 Pages, *LSFM Research Group, Computer Science Department, Science and Engineering Faculty, Laval University Quebec, Canada*

[14] Henrik Theiling: *Extracting Safe and Precise Control Flow from Binaries*, In *Proceedings of the Seventh International Conference on Real-Time Systems and Applications (RTCSA) 2000*

[15] Hans Langmaack: *On a Theory of Decision Problems in Programming Languages*, In *Lecture Notes in Computer Science; Vol. 75; Proceedings of the International Conference on Mathematical Studies of Information Processing 1978*

[16] Sebastian Krahmer: *Generating runtime call graphs*, University of Potsdam, May 2006, ISSN 0946-7580