Lo, the year was 1997 and humanity completes its greatest feat yet—nearly thirty years after NASA delivers the lunar landings, StormCat 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 an 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. 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. 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.

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:

```
# 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 )
    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 77f1eccc 77f1ad23 77f15f60 77f1700a 77f1d969
00454328 00aabbcc 77f17090 77f16f79 00000000 7e429766
0045433c 7e43ee5d 7e41940c 7e44fa5 7e42fbd 7e42aeab
```

---

37: unzip pocorgtfo18.pdf iTetrinet-wiki.zip
38: http://pytrinet.ddmr.nl/
Praise the Stack! We landed inside the import table.

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.

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 backwards 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:

```
sub_424620 shallow jmp short loc_4246C8
```

| 0043F963 | mov ds,:dword_453F28[eax+4], ebx |
| 07202780 | sub_424620 proc near |
| 0720269F | sub_424620 var_20 = byte ptr -20h |
| 072026B0 | sub_424620 Msg = MSG ptr -1Ch |
| 072026C0 | sub_424620 push ebx |
| 072026C1 | sub_424620+1 push esi |
| 072026C2 | sub_424620+2 add esp, 0FFFFFFE0h |
| 072026C3 | sub_424620+5 mov esi, eax |
| 072026C4 | sub_424620+7 xor ebx, ebx |
| 072026C5 | sub_424620+9 push 1 ; wRemoveMsg |
| 072026C6 | sub_424620+B push 0 ; wMsgFilterMax |
| 072026C7 | sub_424620+D push 0 ; wMsgFilterMin |
| 072026C8 | sub_424620+F push 0 ; hWnd |
| 072026C9 | sub_424620+11 lea eax, [esp+30h+Msg] |
| 072026CB | sub_424620+15 push eax ; lpMsg |
| 072026CC | sub_424620+16 call PeekMessageA |
| 072026CD | sub_424620+1B test eax, eax |
| 072026CE | sub_424620+8E lea eax, [esp+20h+Msg] |
| 072026CF | sub_424620+92 push eax ; lpMsg |
| 072026D0 | sub_424620+93 call TranslateMessage << !! |
| 072026D1 | sub_424620+98 lea eax, [esp+20h+Msg] |
| 072026D2 | sub_424620+9C push eax ; lpMsg |
| 072026D3 | sub_424620+9D call DispatchMessageA |
| 072026D4 | sub_424620+A2 jmp short loc_4246C8 |

2765177
X.000789
77124.710663

do it in 6 seconds on your hand-held

Curta Calculator
Compact, quick and simple. The Curta adds, subtracts, multiplies, divides, square roots, takes square roots with absolute accuracy. There is no estimating. It does every thing a calculator 10 times as large and 10 times as heavy can do. And it costs half as much. No wonder that almost every successful rallyist uses a Curta.

It will probably never wear out. Digits are etched and colored white against a matte black finish. No eye strain. Controls and handling surfaces are deeply knurled. Very satisfying in your hand. And we include a metal carrying case.

YOU CAN BUY A CURTA from Burns Industries, the home of Curta Calculators (they’re made for us in Lichtenstein). The cost for the model shown is $11.50. Add $1.50 for shipping. Add $1.00 for delivery to the west coast; $2.00 for delivery to Hawaii or Alaska. Add $5.00 for waiting 4 weeks. We’ll send you a Curta by return mail. Guaranteed satisfaction or your money back. Or ask for our Curta literature.

Burns Industries
36A Delaware Avenue, Buffalo 2, N. Y.
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.

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
2: 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, and the editors of this fine journal have resurrected the abandoned website, http://tetrinet.us/.