20:06 A Short History of TI Calculator Hacks

by Brandon L. Wilson

A lot of people are probably familiar with Texas Instruments graphing calculators from school, those overpriced devices that we were required to buy for math class. Some people are also familiar with the fact that these calculators are programmable, that they can be made to do all sorts of things, such as taking notes or playing games.

But what people outside of the calculator community might not know is that these devices are great learning tools for getting into programming, and even reverse engineering. A big chunk of what we know about programming graphing calculators, we know because we figured it out ourselves. We wrote code not knowing what would happen, we'd run tests, experiment with what the hardware would do, and so on. That's never more true than with trying to break the security built into these things. Why would we want to do that? Well, we'll get into that.

I have way too many calculators. They are what got me started in the software development industry, and because of them, I'm now circling around the security industry.

There are one or two people who have more in terms of numbers, but mine is the largest in that it has at least one of every model ever mass-produced. I have at least one of every model from all over the world, every hardware revision, every color variant, every ViewScreen or teacher's edition, every EZ-Spot yellow school version, as well as a number of one-of-a-kind or near-one-of-a-kind prototypes and engineering samples.

I grew up with these things, they gave me my career and my life. I love them, and I want to make it so they can do absolutely everything they are capable of and then some, and make sure that everyone else can, too, because I'm not the only one. They have jump started a lot of careers, teaching so many of us about low level programming, embedded systems, and hardware and software hacking.

My hope is that I can share with you a little bit of my journey with these devices, how far they've come, and maybe learn a little something or be entertained along the way.

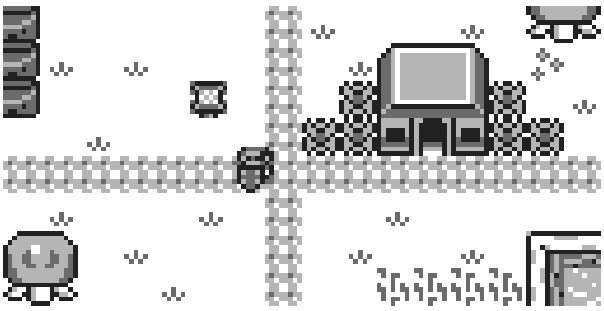
First and foremost, a graphing calculator is a calculator. It's capable of doing everything a scientific calculator can do, but it also has a large screen enabling the graphing of equations, tracing solutions along a graph, drawing, and so on. They even have a 2.5mm I/O port, or in some cases USB, so that you can share variables and programs between calculators, or connect it to a computer and share them with anyone in the world.

They are programmable, which means you can create programs to help you solve math or engineering problems, using a BASIC-like language TI invented called TI-BASIC. It does have some very basic commands for programming games, such as gathering keypress input, but TI-BASIC is just way too slow to really utilize the hardware to its maximum potential.

So for that, we have assembly language. Now, in one form or another, every model, with the exception of the TI-80, is capable of running arbitrary native code. Some of these have this capability built into them, and some of them had to be hacked first, by the graphing calculator user community.

Z80 Models

The first models used the Zilog Z80, a classic processor used in a number of devices. It's a 6MHz, or on some models, 15MHz 8-bit CPU, with 16-bit addressing, meaning it can access a maximum of 64KB of memory at once, and it has an 8-bit I/O port interface, so you can interact with hardware by `out`putting or `in`putting from one of up to 256 logical ports. They have anywhere from 32KB of RAM all the way up to 128KB. And some of them, the most interesting ones, have Flash memory, which ranges anywhere from 1MB up to 4MB.



TI-85, ZShell and the Custom Menu

The first model capable of running native assembly programs was the TI-85, a very old model you don't see these days. Rumor has it that TI employees actually had a bet as to whether we'd figure out a way to run native assembly programs. That was a safe bet, because the community did figure out a way, and it was through something called ZShell.

To explain how ZShell works, we should begin by understanding "Backups" that are transferred by the TI-Graph Link I/O cable, which is what connects these old calculators to a computer. These backups are just dumps of the entire RAM, not just where variables are stored, but the system's RAM as well.

The calculator's operating system also supports something called "Custom" menu entries, which you access with the Custom button on the keyboard. You could add your most commonly used OS commands in there and be able to access them easily.

The way the OS stores things in this menu is by just keeping track of the address of the code that would handle this OS command. And it keeps track of this in System RAM, which is included in the computer backup.

All we have to do for code execution is to change the address of one of these custom menu entries to point to code that we also embed in the RAM backup. That is what ZShell is, just a small program that lets you run other programs which are stored on the calculator in the form of String variables.

TI-82 and Code Execution through Reals

Then the TI-82 came along, and it also had to be hacked to allow execution of native assembly code. It has no Custom menu, so another method had to be found. It does have memory backups, so we began by taking a look at other things that are stored in System RAM.

The TI-OS is essentially just a series of "Contexts," which are kind of like built-in applications, things such as the home screen, the equation editor, the graph screen, etc. Each context has a table of addresses that point to handlers for different things, such as what happens once you press a key. The keypress handler is called the `cxMain` handler, because it's the main, most important handler. Whenever you switch to a new context, these handler addresses are stored in System RAM. Our goal is to find a way, at runtime, to overwrite the `cxMain` handler.

We do this by abusing another feature of these calculators, which is storing values to variables, such as Real variables.[19] These numbers are stored in RAM as nine bytes, and when you copy one variable to another, these nine bytes are just copied from the source variable to wherever the data for the second variable is.

So if we modify one real variable, such as `X`, with the bytes we want, like the address of code we embed in the memory backup, and then modify the location of a second real variable, such as Y, to point to `cxMain` instead of the variable data's real location, then we can overwrite `cxMain` by just storing `X` to `Y`. Once you do that, `cxMain` is overwritten, and the next time you press a key, our code is running! That gets us a shell with which to run other programs, just like on the TI-85.

TI-83 Backdoor, TI-86 Support

Then came along the TI-83, except this model actually has a backdoor in it, put there by Texas Instruments, which allows directly running assembly programs stored in RAM. This backdoor is hidden in the Send(command, which is normally used for transferring variables from one calculator to another via the 2.5mm I/O port. But if you put a 9 right after the command, it won't transfer the variable, it'll instead execute it as native code. The TI-83 is the first calculator I ever had, so this was around the time I joined the calculator community.

When TI saw there was a booming interest in assembly programming through the TI-83 backdoor, they added really nice assembly support to the TI-86, which is a new-and-improved TI-85. This calculator has a brand new command, `Asm(`, intended for running assembly programs right from the be-

[19] They are Real in the mathematical sense, in that they are not Complex.

ginning. TI not only provided some basic documentation for how they use System RAM and how User RAM is laid out, they even included OS hooks so we could integrate with the OS and expand its functionality! It was really quite nice for its time.

A Dozen Models with Flash

And then came Flash technology. These, to me, are the most interesting models, because these are upgradeable, in terms of OS upgrades, Flash applications (which have tighter OS integration and are stored in Flash instead of RAM), USB ports, and security implementations to protect some of this cool new functionality. And whenever something is designed explicitly to keep you from doing something, it's always fun to try to break it.

First off, they made the TI-83, then they made the TI-83 Plus, and then they made the TI-84 Plus, so there was never actually a plain old TI-84. That would be confusing, because that would leave you to believe that because it doesn't have "Plus" in the name, it might not have Flash memory.

But of course, TI did make one model called the "TI-84 Pocket.fr," which is just a physically-smaller TI-84 Plus, it's identical in every way. What's even worse, they made a TI-84 Plus Pocket SE which is just a physically-smaller TI-84 Plus Silver Edition, except they *did* put "Plus" in the name.

And then there are all sorts of duplicates of the exact same calculator, just with a different name on it. You have the TI-82 Stats and TI-82 Stats.fr, which are really just TI-83s, you have the TI-83 Plus.fr which could actually be referring to two different calculators, one is just a TI-83 Plus and the other is a TI-84 Plus Silver Edition.

And then the TI-82 Plus, which is just a TI-83 Plus, and then the TI-83 Premium CE, which is the same as the TI-84 Plus CE, and then the TI-84 Plus T, T for "test," but that's actually a TI-84 Plus Silver Edition.

Motorola 68K Models

While the Z80 models are by far my favorite, there are also a number of Motorola 68K models. These began with the TI-92, which came out around the same time the TI-85 did. It has a QWERTY keyboard, which is neat but gets it banned from most standardized tests. If it as a keyboard, it's a computer, they say.

One thing that's unique about this model is that it has an expansion port on the back, which would let you add features or even turn it into a different model entirely. There's the TI-92 II module and the TI-92 E module, E for Europe, that essentially just added more RAM and language options. And then there's the TI-92 Plus module, equally as rare but way more interesting, as it turns it into a TI-92 Plus, giving it Flash memory and upgradeability. That model is basically the same as the TI-89, except the TI-89 doesn't have a QWERTY keyboard.

And then came the TI-89 Titanium, which has some minor hardware changes and most noticeably adds a USB port.

NSpire Models (ARM)

There's also the TI-Nspire models, which use ARM. I hate these calculators because they're big and bulky, and they were clearly designed for students and not for engineers. But they do have swappable keyboards, and probably the most significant one there is the TI-84 Plus keypad, which causes it to emulate a TI-84 Plus, making it kind of sort of useful again. There are versions that don't have a Computer Algebra System (CAS), and versions that do.

Then came the TI-Nspire CX models, again both CAS and non-CAS versions. These have color LCDs and are redesigned to be a little sleeker, so they're alright, I guess.

Another big reason to hate these guys is that they are completely 100 percent locked down, with no way to execute native code at all. Unless you use Ndless, which is, for lack of a better term, a jailbreaking utility along the lines of ZShell. For some reason, TI fights this really hard. They fix vulnerabilities that Ndless uses as soon as possible, way faster than with the other models.

The eZ80 and its Flat Memory Model

And then we have the eZ80 models, the newest models that have color LCDs. Unlike the Z80 models, these use an eZ80 CPU with 24-bit addressing and backward compatibility with Z80 code. The ASIC and hardware interface is completely new, totally redesigned with security in mind. Unlike the Z80 models which use a paging or bank-switching system, the eZ80 models have a flat memory model, which will be interesting later on.

The TI-83 Premium CE, hardware-wise, is identical, but has a different OS on it which includes an

exact math engine and is only sold in Europe. TI really wants to prevent being able to run this nicer OS on the US TI-84 Plus CE, but as we'll see, they're not going to succeed in that.

And then finally the TI-84 Plus CE-T, which is simply the European version of the TI-84 Plus CE.

So having said all that, there are some really cool things you can do that have nothing to do with calculators, or math, or school. Since some of these models have On-the-Go USB ports, it is possible to connect any number of USB peripherals to it, anything from Bluetooth and WiFi adapters so calculators can communicate wirelessly with each other, to serial adapters, to keyboards and mice, even USB flash drives, hard drives, and floppy drives, all of which exist.

These calculators have a unique USB On-the-Go controller, one that's flexible enough to allow real abuses of the protocol. Probably the best example of that is when the PlayStation 3 jailbreak first came out, shortly after OtherOS was taken away.

Well, long story short, it was a USB-based exploit that required connecting a Teensy or similar device to your PS3 to enable unsigned code execution. Of course Teensy's all over the world quickly sold out.

So I looked into how it worked and realized that it essentially simulated a USB hub, then virtually attached and detached a bunch of fake devices in order to arrange the heap for a memory corruption exploit. In order for that to work, the USB peripheral has to be able to pretend to be other USB devices by changing its own device address in software, and that is something the calculators are able to do. After I ported the exploit, people were able to jailbreak their PS3 using a graphing calculator.

You can simulate other USB devices as well, such as the USB portal used with RFID video games like Skylanders, Disney Infinity, Lego Dimensions. I've even booted a PC off the calculator by having it pretend to be a USB Mass Storage device!

Why have Security in a Calculator?

Why does TI bother to secure their calculators? Well, when Flash memory first came into the calculator world, they sold Flash applications for seven to fifteen dollars apiece. These applications included a pocket organizer, spreadsheet applications, a periodic table and enhancements to the built-in math capabilities. They even published games.

They provided an SDK for free, but charged a hundred dollars for the right to release three Flash applications in their online store. Naturally, they wouldn't want these applications to be pirated, so they had to restrict how and where these applications get installed.

They also want to prevent cheating in the classroom, by locking down the calculators further during tests and exams.

All of this depends upon preventing tampering of the operating system, where we could easily disable or defeat their security mechanisms. In fact, I'm convinced we could make a better OS than them in terms of math capabilities and performance.

There user community, of course, wants to maintain control over the overpriced hardware that we own. There are countless numbers of things we can make these devices do which not only help the calculator community.

Now that we know a little bit about who the players are, let's get back into the technical aspects of how these calculators work, and how the security is implemented in them, and how we can, have, and will continue to defeat it.

The First Z80 Flash Vulns

At a hardware level, the Z80 models really consist of three things: the ASIC, the Flash chip, and then all the other hardware that the ASIC interacts with, such as the LCD display, the USB and serial I/O ports, and the keyboard.

Now, this is not completely accurate as the hardware has changed over the decades. For example, the RAM wasn't always internal to the ASIC, and neither was the CPU, but this is the most common configuration you would likely come across today.

As I mentioned, the Z80 is a 6MHz CPU with 16-bit addressing, so it can only access 64KB of memory at one time. They use bank switching, where that 64KB is split up logically into four 16KB pages, also called banks. Each of these banks can hold any 16KB region of memory you want, so if what you want to access isn't currently swapped into one of the banks, you just reconfigure that bank to point to the 16KB you want, and there it is.

As far as accessing the hardware, the Z80 has 8-bit I/O addressing, so there's a maximum of 256 I/O ports it can talk to. The purpose of each I/O port is different for each model, but the Flash models all follow the same basic pattern, which is everything from port `0x00` all the way up to `0xAF`. These do

everything from ASIC configuration, LCD access, keyboard input, USB control, everything.

Z80 Memory Banks

Bank	0	1	2	3
Base Addr	0000	4000	8000	C000
Port		06	07	(05)
	ROM Page 00	Any ROM Page	Any ROM Page	Any RAM Page
or				
	ROM Page 7F	Any RAM Page	Any RAM Page	

There are a few rules about how the bank switching works in the 83+ and 84+ series. As I said, it's split up into four banks of 16KB each, starting at `0x0000`, `0x4000`, `0x8000`, and `0xC000`.

The first bank, except for some weirdness during cold boot, always has ROM page `0x00`, which is the start of the OS. The second bank is used to swap in different chunks of the OS, which is way bigger than 64KB, constantly swapping in what it needs when it needs it.

The third and fourth banks typically have RAM pages swapped in, meaning there's usually 32KB of RAM accessible to the OS at any given time. Some of that is User RAM, and some of that is the hardware stack, and then the rest is system RAM that the OS can use internally.

And as you can see, the last three banks all have I/O ports that control what page is swapped in. If you want to swap ROM page `0x01` into the second bank, you write a `0x01` to I/O port `0x06`. Or if you want to swap RAM page `0x81` into the third bank, you write `0x81` to I/O port `0x07`.

By far the *most* important I/O port in the entire ASIC is port `0x14`, which controls Flash unlocking and relocking. Whenever the Flash chip is locked, which is almost always the case, write and erase commands to the Flash chip are ignored. So essentially, you cannot modify Flash until you unlock it. It also controls whether certain I/O port values can be modified. We call that a "privileged" I/O port, because Flash has to be unlocked before you can write to it. So it doesn't deal with just Flash, that's just what it's come to be known by.

How port `0x14` works is very simple; you write a `0x01` to unlock it or a `0x00` to lock it back. What's not simple, though, is when code is allowed to write to that port. A special sequence of Z80 instructions has to be fetched and executed from a "privileged" Flash page before writes to port `0x14` will stick. And it's no coincidence that the unlock sequence contains instructions like `IM 1` (interrupt mode 1) and `DI` (disable interrupts) to explicitly prevent interrupts from interfering with this process.

The privileged page ranges are mentioned there, but as you can see, the only pages allowed to modify Flash are the OS and boot pages. So you can't modify the OS unless you are the OS or the Boot Code. That leaves us out of luck for unlocking it ourselves.

Tricks that Almost Work to Unlock Flash

To give an example with how TI uses this protection, here's the logic behind receiving and installing an OS upgrade. In a loop, the Boot Code will 1) receive a chunk of OS data and where it should be written to on the Flash chip, 2) unlock Flash using that privileged sequence and writing `0x01` to port `0x14`, and then checks for a bunch of tricks we might use to steal control away while it's unlocked, 3) write the OS data to the specified area of the Flash chip, and then finally 4) relock Flash back using the same privileged sequence as before, writing a `0x00` to lock it back.

Anytime the OS does something involving modifying Flash, it will unlock it, perform some simple operation as quickly as it can, and then relock Flash.

I mentioned it checks for trickery. Specifically,

- It checks to make sure that SP, the stack pointer, lies between `0xC000` and `0xFFF8`. It does this to make sure SP is pointed to somewhere in RAM, so that when it returns back to the caller, it can get what it assumes would be a valid return address from the stack.
- It checks to make sure port `0x06` contains a privileged Flash page, because that's where any Flash unlocking code would be running from.
- It checks port `0x07` to make sure it contains RAM page `0x01`, which is where System RAM is and what the OS considers the normal scenario.
- It complements the bytes at `0x8000` and `0xC000`, which confirms that the third and fourth banks contain writable RAM pages. I'll attempt to illustrate why it does this.

If only the SP were in ROM

Why would TI care if we point SP, the stack pointer, to an area of Flash? Well, let's play this out.

For starters, modifying Flash is complicated. It's not as simple as loading a register value to a memory address. It requires a sequence of memory-mapped commands, commands like Get Chip ID, Erase Sector, Program Byte, and so on.

If we point SP to a location that's definitely in ROM, such as `0x1000`, which is deep in ROM page `0x00`, and then jump into some code that unlocks Flash and calls a subroutine, something interesting happens.

The CALL instruction is going to attempt to write the return address to the location pointed to by SP, but because SP is pointing to ROM, a bunch of `0x80` bytes in this example, those writes are going to be ignored. So when it finally encounters a return instruction, it will read the two bytes pointed to by SP, which is `0x80` and `0x80`, and it'll jump there, to `0x8080`. Not at all what the code intended to do, but because we messed with SP, that's exactly what happens.

So this would be a really cool way to steal control away from the OS and Boot Code, but no, they did think of that. So what next?

Executing Misaligned Instructions

Through experimentation, we eventually learned that the privileged sequence of instructions only needs to be read from the privileged page; it *doesn't* have to be executed. This requires thinking about what actually happens on the data bus when instructions are being executed.

When it goes to execute the "RLC (Rotate Left with Carry)" instruction, it first has to read the bytes that make up that instruction. Because it uses index register `IX`, that's a four byte instruction, so it reads `DD CB 00 00` from the privileged page. Then it has to actually execute that instruction, and to do that, it has to read the byte at `IX` at offset `0`. That is the `0xED` byte from the privileged page. Then it goes to execute the "load HL into D" instruction, which means it has to read that opcode, which is `0x56` from the privileged page. Then it actually executes it, which means it reads the `0xF3` byte from the privileged page.

The Z80 equivalent of all those bytes is, coincidentally, "`nop; nop; im 1; di`," which is the unlock sequence.

The big advantage here is that this does NOT require actually executing the `DI` (Disable Interrupts) instruction or the `IM 1` (Interrupt Mode 1) instruction, which means we could use an interrupt to steal away control.

So all we need to do is find the instructions on a privileged page; unfortunately, those are nowhere to be found. So as awesome as this would be, we cannot use it.

Port 0x05 Swaps the Call Stack's Bank!

Well, here comes along the TI-83 Plus Silver Edition, which is an enhanced version of the TI-83 Plus. It has 128KB of RAM instead of just 32KB, it has a Flash chip twice as large, and its CPU is capable of switching between 6MHz and 15MHz. Its ASIC got a few upgrades as well, namely I/O port `0x05`.

This I/O port actually allows controlling the RAM page swapped into the last bank, something that couldn't be done on the original TI-83 Plus. The thing is, TI didn't update their Flash unlock trickery checks to also validate the value of port `0x05`. This can be used to our advantage.

The OS always expects RAM page 0x01 to be in the third bank, and RAM page 0x00 to be in the fourth bank. But what happens if we swap the same RAM page into the last two banks?

Bank	0	1	2	3
Base Addr	0000	4000	8000	C000
Port		06	07	05
	ROM Page 00	ROM Page 7C	RAM Page 01	RAM Page 01

Now things are all kinds of screwed up. Even though SP, the stack pointer, is pointing to the last bank, the stack is most certainly not there anymore.

In fact, we have the same page swapped into two banks at the same time. If I were to write a value to the first byte of the third bank, I would actually be able to read it from the first byte of the fourth bank! That's definitely very interesting.

What we need is to find a section of the OS, or Boot Code, that unlocks Flash, writes a value to the third bank, and then attempts to relock Flash back. As luck would have it, there's a very convenient block of code that does that. There is a particular bit, and in fact an entire byte, of the certificate region of Flash that holds whether the OS is valid or not. If it's valid, as it usually is, the value will be 0x00.

What we can do is jump directly into the Boot Code at the point that it unlocks Flash, just before it reads this byte from the certificate. It will read it and store it to an area of System RAM called OP1, which is in the third bank, at address 0x8478.

Since we have just used I/O port 0x05 to swap RAM page 0x01 into both of the last two banks, writing a zero to 0x8478 will also write a zero to 0xC478, which is exactly 16KB ahead, in the fourth bank.

If we craft things just right, we can set SP so that by the time it gets to the write to 0x8478, SP will be pointing to 0x8478. When it performs that write, it will corrupt the return address that SP is pointing to.

If the return address used to be 0x46E1, writing that zero has changed it to 0x00E1. So as soon as the code hits the return instruction, it's not going to return to the Boot Code. It's going to return to 0x00E1 instead, which is deep in the OS interrupt, in ROM page 0x00. We can use an OS cursor hook at that point to steal control away, clean up the stack and restore the value of port 0x05, and we have Flash still unlocked, ready for us to use!

Universal Flash Unlock Exploit

That's great and all, but this port 0x05 trickery only works on the TI-83 Plus Silver Edition and up. The original TI-83 Plus has no port 0x05, so it isn't vulnerable to this bug.

Even worse, we had to use an OS hook to steal control back, we had to hard-code the value of SP based on the call stack, and we had to hard-code a return address that starts with 0x00, all of which could change between OS and Boot Code versions.

What would be really nice is if we had something that worked on every hardware revision of every model in the family, independent of the OS and Boot Code versions. To do that, we're going to have to attack functionality that not only exists on all models, but isn't likely or even able to be changed easily.

One such feature is the OS' ability to receive Flash applications from a connected computer or another calculator. Since Flash applications are fixed multiples of 16KB in size, even the smallest Flash application cannot fit in RAM all at once. That means the OS *must*, in a loop, receive a chunk of Flash application data, unlock Flash, write that chunk to an *arbitrary* location in Flash, and then relock Flash back, over and over again until all of the application is received and written to Flash. This has existed in every OS version for every model since the beginning, and they cannot take it out, so if possible, it's the perfect thing to attack.

Before jumping into the OS code that unlocks Flash and writes data to an arbitrary destination, we know we have control over the destination Flash page and address, the number of bytes to write, and the bytes to be written, but we don't have control over the source address, which is in RAM. That means bit 7 of H will always be set, and bit 1 of iy+25h will remain reset. If we could set it, then the code that wraps DE from 0x8000 back around to 0x4000 will not run, and this routine will write data to an address above 0x8000, which is all RAM. So it would effectively turn this command into a RAM-to-RAM copier.

That's actually a good thing, because we can use this to overwrite the data near SP, the stack pointer, to all the same value, such as 0x8080. When this routine hits a return instruction, it will jump to

```
.nolist
#include "ti83plus.inc"
.list
.org userMem-2
UnlockFlash:
;Unlocks Flash protection.
;Destroys: appBackUpScreen
;          pagedCount
;          pagedGetPtr
;          arcInfo
;          iMathPtr5
;          pagedBuf
;          ramCode
        in a,(6)
        push af
        ld a,7Bh
        call translatePage
        out (6),a
        ld hl,5092h
        ld e,(hl)
        inc hl
        ld d,(hl)
        inc hl
        ld a,(hl)
        call translatePage
        out (6),a
        ex de,hl
        ld a,0CCh
        ld bc,0FFFFh
        cpir
        ld e,(hl)
        inc hl
        ld d,(hl)
        push de
        pop ix
        ld hl,9898h
        ld (hl),0C3h
        inc hl
        ld (hl),returnPoint & 11111111b
        inc hl
        ld (hl),returnPoint >> 8
        ld hl,pagedBuf
        ld (hl),98h
        ld de,pagedBuf+1
        ld bc,49
        ldir
        ld (iMathPtr5),sp
        ld hl,(iMathPtr5)
        ld de,9A00h
        ld bc,50
        ldir
        ld de,(iMathPtr5)
        ld hl,-12
        add hl,de
        ld (iMathPtr5),hl
        ld iy,0056h-25h
        ld a,50
        ld (pagedCount),a
        ld a,8
        ld (arcInfo),a
        jp (ix)
translatePage:
        ld b,a
        in a,(2)
        and 80h
        jr z,_is83P
        in a,(21h)
        and 3
        ld a,b
        ret nz
        and 3Fh
        ret
_is83P: ld a,b
        and 1Fh
        ret
returnPoint:
        ld iy,flags
        ld hl,(iMathPtr5)
        ld de,12
        add hl,de
        ld sp,hl
        ex de,hl
        ld hl,9A00h
        ld bc,50
        ldir
        pop af
        out (6),a
        ret
.end
end
```

Universal Unlock Exploit for the TI 83+ Family

`0x8080` instead, where we can take control, clean up the stack, and return with Flash still unlocked.

So how can we ensure bit 1 of `IY+25h` is set even when this routine will start out by resetting it?

If we point `iy-25h` to a point in Flash where bit 1 is set, then the Boot Code's attempt to reset it with the `res` (Reset Bit) instruction will not work. If you remember, modifying Flash involves memory-mapped commands to program one byte at a time, so the `set` and `res` instructions will have no effect. See page 39 for a working example.

Now, this is all entirely dependent on the fact that they never set `iy` after Flash is unlocked, so it's fixed easily enough in the OS. But similar functionality exists in the Boot Code, and that can't be easily fixed, certainly not on existing hardware. And even if they did fix it, there are a number of other Flash unlock exploits that can be used. I have about a dozen different methods that I've never disclosed, just in case TI ever starts to get aggressive with fixing these things.

RSA Key Factoring

Being able to unlock Flash and modify it ourselves is nice, but if we wanted to write our own OS, we'd have to rely on custom OS receivers, which are platform-dependent, error-prone, and just troublesome to mess with. It would be nice if we could just patch the OS and re-sign it ourselves, or write our own OS and sign it, with TI's private RSA key. But of course, they aren't going to just hand that key over to us.

Flash-upgradeable Z80 models started around the time that the TI-73 came out, and that was around 1997. And in 1997, 512-bit RSA keys were looking pretty secure. If you don't know, RSA's strength is in the inability to factor the public key, which is an extremely large number, into two prime numbers. And computing power not being what it is today, that was considered impossible at the time.

But, flash forward ten years or so, and one person decided to give it a shot anyway on his computer. He used something called the General Number Field Sieve, which, at least at the time, and maybe still so, was considered the fastest and most efficient known method of factoring numbers into primes. He kicked off the process for the TI-83 Plus OS signing key and let it run on his computer for two months or so before it finally spit out the primes. He had proven what was long disregarded, that it was possible to factor these keys. So he posted about it online, and very shortly after, TI silenced him.

They actually sent someone to his home to talk to him, to strongly encourage him not to work on this anymore, not even to talk about it. As you can imagine, this scared the crap out of him.

But, the damage was done, and the community knew what was possible. They took the remaining thirteen public keys and started a BOINC distributed computing project to factor the rest of them. We had hundreds, thousands of people all helping to factor the keys as quickly as possible, and before we knew it, we had all thirteen private keys in just one month, all under TI's nose and without them finding out.

Since no one ever had the OS keys, or even the application keys on most models, there were no tools to sign modified OSes or applications. I threw some together, validated that every single key was correct and could produce OSes and Flash applications that each calculator would accept, and published those tools along with the key files needed to use them.[20] That seemed to be the final straw for TI, because they sent me a DMCA takedown notice.

Were it not for the EFF, the Electronic Frontier Foundation, stepping in and offering to defend me legally against TI's threats, I would've been forced to comply. The EFF sent a letter to Texas Instruments stating that it isn't possible to copyright a number, which is essentially what I published, and that they should leave me alone because it isn't worth destroying a person over. TI did not respond to that letter, so the matter was dropped, and I'm still hosting the 512-bit keys to this day.

Knowing that they had lost this particular battle, TI started using impossible-to-factor 2048-bit RSA keys in newly-manufactured models of the TI-84 Plus ad TI-84 Plus Silver Edition. Since the hardware was never designed to validate such a large signature, validating the OS now takes six minutes! This is simply unacceptable, so we'll have to fall back on Flash unlock exploits again to undo this.

[20] `unzip pocorgtfo20.pdf ti83pluskeys.zip`

Defeating the 2048-bit Signature; or, John Hancock Corrupts the Call Stack

So to get rid of this six minute validation, we have to understand how the calculator boots and how OS upgrades work.

When first turning the calculator on, the Boot Code is the first thing to get control. It does some basic hardware initialization, then checks the OS valid marker stored on sector 0 of the Flash chip. If that marker is valid, it jumps into the OS, and the calculator starts normally. If that marker is NOT valid, then it waits to receive a new, valid OS over one of the link ports.

For a typical OS transfer, the first thing the Boot Code will do is invalidate the OS both in the certificate and by erasing Flash sector 0, which will reset the OS valid marker. In a loop, it keeps receiving small chunks of the OS over and over into RAM, and then unlocking Flash, writing that to its destination, and then re-locking Flash. Once that's all done, it's time for the Boot Code to validate the 512-bit signature in the OS, which is effectively useless now because we can generate that signature ourselves. Then, it goes to validate the 2048-bit signature. And if all those checks pass, it marks the OS as valid in Flash sector 0 and the certificate, and then it jumps into it.

Digging in a little further, let's look at how it validates this 2048-bit signature. Unlike the original 512-bit signature, this new one is stored length-indexed, meaning that there's a word at the beginning indicating it's 256 bytes. If you know the signature is 2048-bit, or 256 bytes, why store the length? It opens up the possibility that it could be exploited, and as it turns out, yes, they don't bounds-check this length, so we can take advantage of it.

We can embed a *really large* signature into the OS update. Because the Boot Code doesn't check that it's a sane value, it will blindly copy the signature to the start of RAM, at `0x8000`. So we can store `0x80` bytes of garbage there, then a Z80 jump instruction, which is opcode `C3` followed by the address. Then we can put lots and lots and lots of `0x80`s that eventually will totally overwrite RAM including the stack.

The next time the code tries to return, it returns to address `0x8080`, where we have a jump to where we calculated the payload would really be at.

Once we get control, we can do some cleanup, such as marking the OS as valid both on Flash sector 0 and in the certificate, and then just jumping to the start of the OS.

The nice thing about this technique is that no custom OS transfer tools are required. We just create a specially-crafted OS upgrade file. Better still, this exploits the read-only Boot Code, so all models manufactured so far are vulnerable.

Patching the 84+ Boot Code

Another big discovery in the community, and another nail in the coffin on the security of the TI-83 Plus and TI-84 Plus series, has to do with modifying what should be read-only boot sectors.

One thing I noticed is that the TI-84 Plus and TI-84 Plus Silver Edition boot sectors are *almost* identical. In fact, other than the fact that the first one has a 1MB Flash chip and the other one is 2MB, they *are* identical calculators in every way, except for one little I/O write.

When the calculator is first booting and initializing hardware and I/O, it writes either a `0x00` or a `0x01` to I/O port `0x21`. Now, this is a protected port, which means Flash has to be unlocked before it can be written to. But, both calculators run exactly the same OS, which reads the value of port `0x21`, bit 0 specifically, to determine which model it's running on. It's critical that it know this, for a very important reason: the OS is actually organized into two sections.

The Flash layout for the TI-84 Plus is on page 42. It has `0x40` Flash pages. The first OS section is at the very beginning of the Flash chip at sector 0, and it runs from Flash page `0x00` to page `0x08`. Near the end of the Flash chip is the second part of the OS; these are the privileged pages. Both the upper OS page range and the boot page are privileged, but the boot page is supposedly read-only.

And then in between the two OS sections is the user archive, where Flash applications, archived variables, and so on are stored.

The Flash layout for the TI-84 Plus Silver Edition is basically the same, except that the Silver Edition has a Flash chip that's twice as big. The boot page is now `0x7F` instead of `0x3F`, and the upper OS page range is `0x7C` and `0x7D`, instead of `0x3C` and `0x3D`.

The Boot Code initially sets the value of I/O port `0x21`, indicating which model it is, but what would happen if we unlock Flash and modify it ourselves? If a TI-84 Plus Silver Edition writes a `0x00` to port `0x21`, then the OS would believe it's actually a TI-84 Plus non-Silver Edition, and vice versa.

TI-84+ Flash Layout (Non-Silver Edition)

Flash Pages	User Archive	Flash Pages	Flash Page
0x00 to 0x08	Flash Apps	0x3C to 0x3D	0x3F
Lower OS	Archived Vars	Upper OS	Boot Page
		Privileged	Privileged
			Read Only

TI-84+ Flash Layout (Silver Edition)

Flash Pages	User Archive	Flash Pages	Flash Page
0x00 to 0x08	Flash Apps	0x7C to 0x7D	0x7F
Lower OS	Archived Vars	Upper OS	Boot Page
		Privileged	Privileged
			Read Only

Now, normally this would just crash the calculator, because it would suddenly be looking at page 0x3C, for example, when what it really wanted was 0x7C. But, I had an idea that I could just copy the upper OS pages and the boot page to the middle of the Flash chip, from pages 0x3C to 0x3F. So, when the OS went to look for page 0x3C, it would actually find it, and it would continue to function normally. That effectively cuts the user archive in half. So that was my thought, I could force the OS to only think half the user archive was there.

But, when I tried to put this into practice by changing port 0x21 and copying pages 0x7C through 0x7F to 0x3C through 0x3F, the copy operation wouldn't work. It turns out, there's a really good reason for that.

When I changed the value of port 0x21, I changed which range was read-only! By changing the value of port 0x21, I actually changed the protection from one region to another. So all this time, we thought the Flash chip itself was edit-locked on the boot page, but no, it was the ASIC's port 0x21 keeping it edit-locked. By temporarily flipping the value of port 0x21, we can actually modify the Boot Code!

To write to page 0x7F on an 84+SE, we just write 0x00 to I/O port 0x21, effectively making it temporarily not a Silver Edition. Then we perform the Flash sector erase and write while the page is unprivileged, then restore port 0x21's value to 0x01, making it an SE again.

On the 84+, we do the same thing in reverse by writing 0x01 to port 0x21 to make it temporarily a fake SE, then overwriting the Boot Code page at 0x3F while it is no longer protected!

This made it possible to modify the Boot Code, and modify it we did! We made diagnostic utilities and embedded them in the boot page so that it was impossible to permanently brick it, and–most importantly–we can simply patch out the 2048-bit signature check.

Naturally, when they figured out we could do this, they changed the way the calculators were manufactured. They now edit-lock the boot sector on the Flash chip, so the ASIC protection is redundant.

The 84+ Color Silver Ed Uses Our Bug!

Here's the really fun part: Shortly after, TI came out with their first and only calculator to have a color LCD and the classic Z80 architecture. Not only did it have a color LCD, but it had a 4MB Flash chip instead of 2MB, and they called it the TI-84 Plus C Silver Edition, C for color. That's the only difference between it and the older models. They even used exactly the same ASIC, even though it wasn't designed to work with a Flash chip beyond 2MB.

The problem is, the 4MB Flash chip has a different sector layout compared to the 1MB and 2MB Flash chips used in earlier models. The supposedly read-only boot pages at the end of the 2MB Flash chip are now in the middle of the 4MB Flash chip, which is part of the new calculator's user archive. So in other words, TI now needs to write to the pages that the ASIC is designed to protect. So what did TI do?

They used our workaround! They temporarily toggle which region is protected, because they can't just turn it off, all they can do is misconfigure it a different way, do their writes, and then toggle it back.

Did they get the idea from us?

New Protections of the TI 84+ CE

The constant toggling of port `0x21` actually slows the calculator down too much, so they dropped the TI-84 Plus C Silver Edition in favor of the TI-84 Plus CE, a brand new color calculator with an eZ80 CPU.

The eZ80 sports Z80 backwards compatibility, so it can run regular old Z80 instructions in addition to the new eZ80 ones, which support 24-bit addressing and a 16-bit I/O range instead of just an 8-bit one. Since they now have 24-bit addressing, they ditched the paging and bank switching model in favor of a flat memory model.

TI-84+ CE Flash Regions

Start	Length	Name	
`0x000000`	`0x200000`	Boot	Priv, RO
`0x200000`	Varies	OS	Priv, Writable
Varies	Varies	User	

They also revamped the port protection, since there are no "privileged pages" anymore. Now, certain address ranges are considered privileged. And certain I/O ports, mainly any where the high byte is `0x00`, are considered protected and can only be written to from a privileged address range.

The Boot region at the start of the Flash chip is read-only and always privileged, and then the variable-sized OS follows it. The rest is the User archive. Since the size of the OS can vary from version to version, the ASIC has to be configured at runtime to know which parts of the Flash chip to consider privileged. That range is configured via protected I/O ports `0x001D` through `0x001F`, which can only be modified by code in the privileged region. So how do the protected I/O ports work?

Well, with any privileged I/O port write, TI must load a constant value into a register, write that register value to the protected I/O port, and then immediately verify that register contains the same constant value they just loaded. They have to do that because otherwise, we could just jump into the Boot Code right before the port write with our own value. That's tedious, but they have a bunch of macros to do this kind of stuff for them.

The problem, though, is that the OS size is variable, not constant. It's not something they can hard-code. So, we could set our own register value and jump into the Boot Code right before the port `0x001D` I/O write. Then, we could steal control away through a variety of means, interrupts, whatever.

eZ80's Backward Compatibility Can Bite

The eZ80 has backwards compatibility for running code in Z80 mode. (The native eZ80 mode is called ADL mode.) Even better, any individual instruction can run in ADL mode or in Z80 mode. In ADL mode, you can call a subroutine that runs in Z80 mode, and when it returns, you're back in ADL mode. And even better than that, in that Z80 mode subroutine, you can have ADL instructions such as those 16-bit port OUT and IN instructions.

It's all very convenient, so surely the protection on the protected I/O ports works in both ADL mode and Z80 mode, right? No, no it doesn't.

To effectively negate the protection, we just set the upper bounds of the privileged range to be really high, something like `0xFE0000`. On line 28 of this example, we temporarily jump into Z80 mode to execute a single instruction, one that writes to the protected ports `0x001D` through `0x001F`, which really should not work, and then returns back to the eZ80 ADL mode.

```
OpenAllPortAccess:
    ld a,0FEh
    ld hl,0000h
WriteAccessPortAHL:
    ld.sis bc,001Dh
WriteBCPortAHL:
    push af
    ld a,l
    call DoProtectedWrite
    ld a,h
    inc bc
    call DoProtectedWrite
    pop af
    inc bc
DoProtectedWrite:
    di
    push bc
    push hl
    push de
    ld hl,do_protected_write
    ld de,RAMstart
    ld bc,(do_protected_write_end
          - do_protected_write)
    ldir
    pop de
    pop hl
    pop bc
    jp.sis 0000h
do_protected_write_finish:
    ret
do_protected_write:
    out (c),a
    jp.lil do_protected_write_finish
do_protected_write_end:
```

Someone over there really should have caught this. These new models are less secure than the ones from twenty years ago, and they were trying to *improve* upon that security. In my opinion, the original unlock protection used on the TI-83 Plus and TI-84 Plus series would have worked, so long as they stay on top of code-related exploits. (They didn't, of course.)

So far as this protection goes, the I/O port protection is likely in the ASIC just like before, and can't be fixed through software updates.

Recalling how they used the awkward `0x21` workaround in the TI-84 Plus C Silver Edition rather than patch the ASIC, this ASIC bug is likely here to stay. But, just in case it's not, there *are* other ways.

An Old Exploit for the TI 82 Advanced

To bring things full circle, there is a new model in Europe called the TI-82 Advanced, which in hardware is really a TI-84 Plus non-Silver Edition without the 2.5mm I/O port.

This model is very locked down compared to the others. No more assembly program execution; no more Flash applications transferred from a PC. The only applications are built into the OS, and they put an LED that blinks during tests or exams in place of the 2.5mm I/O port.

So how might we hack this thing? Well, the obvious thing is to resort to the original TI-82 hacks, whose OS even after all these years is still pretty similar to this one.

RAM backups, perhaps? Well, that's normally something that happens over the 2.5mm I/O port, which we no longer have. But, unbeknownst to most people, RAM backups actually do work over USB, sort of. No link software supports it, because we never really bothered to look, but code to handle it is implemented in the OS.

I came up with a specially-crafted memory backup with corrupted Real variables, as well as a script to transfer this memory backup from a PC, and it does work, you can get code execution on it and even unlock Flash.

Then they made a new model, the TI-84 Plus T, which is just the Silver Edition version of this TI-82 Advanced, except they removed the backup functionality from it. So that functionality may disappear soon from the TI-82 Advanced as well, and we'll need a new way in.

Where do we go from here?

What's next? Well, there are still plenty of exploits to release. Ndless for the TI-Nspire is constantly being fought by TI, so help is always appreciated there, and just explained, we need a new method of privileged code execution for the TI-82 Advanced that will work on the TI-84 Plus T. That's kind of an old school challenge that's still outstanding, and I'm sure a clever reader could finish it off with a few weekends of coding.

And then of course there's the TI-84 Plus CE family, where we need to stay on top of new developments, new hardware revisions, new OS versions. You never know when TI is going to make a manufacturing change or an OS update that has a big impact on the community. More than once I've seen them release OS updates that have very serious bugs in them that mess up programs that have been around for decades. If we don't let them know the technical details of what went wrong and how to fix it, who will?