Gary Stark

This document describes the various parts of the Warrior Security Scheme, and the modules affected by it. It also lists specific steps that must be taken in order to fully implement the scheme and gives suggestions on how to make it most effective.

## The Modules Affected

### INT.ASM

Near the top of INIINT, there are two new subroutine calls: init_helper() and init_sloop(). These references were also added to the XREF statement near the top of the file. These routines are described later on.

### SLOOP.ASM

This module contains two new routines: paged_data() and init_sloop(). It has its own section name "sloop" so that it can be located in a specific place in the address space. THIS IS CRITICAL!! The SLOOP chip has been set up with the FREEZE_OFF and FREEZE_ON address strobes such that FREEZE_OFF is triggered by the start of paged_data(), and FREEZE_ON is triggered after paged_data() has done its magic. BE VERY CAREFUL of making ANY changes to paged_data() that might affect this situation.

paged_data() sends a command to the ASIC65 requesting a particular chunk of data. The ASIC65 knows what page (if any) the data is on and returns a series of eight 16-bit indexes that are then used to strobe the SLOOP accordingly. If the requested data is not in the SLOOP area, the ASIC65 may choose to send data that will cause a random page switch. It also will arbitrarily elect to change the SLOOP from BOUNDS_ON to BOUNDS_OFF mode, and vice versa. In all cases, the ASIC65 keeps track of what page the SLOOP is on and what mode it is in. paged_data() then uses the buffer of eight values as indexes off of base address 0x78000, summing the data at those locations in order to strobe the addresses. After the address strobing is done, but before the return, the PC will move through the FREEZE_ON address locking the state of the SLOOP until paged_data() is again called.

init_sloop() works in a similar manner to paged_data() receiving a buffer of nine 16-bit indexes that can be used to initialize the SLOOP to state A0 from any state. It does not pass a parameter to the ASIC65.

### SCROLL.ASM

This module contains the two routines scroll_pf() and floor_pf(). It has been modified in two ways. First of all, floor_pf() now calls paged_data() before reading each playfield floor section, and then calls paged_data() after reading each section to return the current segment data page.

The second way that the routine has been modified is that the code for these routines is acutallly generated by a macro. This way, I have been able to generate 4 identical copies of the two routines, and put each pair into its own named section (scroll0, scroll1, scroll2, scroll3). As described later, each of these sections is located at the start of every SLOOP page. The upshot of all this is that these routines can be accessed no matter what page is currently active. Furthermore, it is likely that the pages will actually switch during the execution of floor_pf().

SECRET.ASM
---

This module contains 3 routines: sum, ramhit, and stack_bomb. These routines were initially authored by Dennis Harper. Currently none are being called, and the whys and wherefores of calling them are discussed later.

sum is a general checksumming routine that returns a true or false value. Also, in order to compile properly the locations page_chk and page_chkend have been given bogus labels internal to this module. Once the target for this routine's checksumming has been established, these need to be made XREFs as indicated in the comment block near the top of the module. Also the equate SUM1 needs to be determined by trial and set appropriately.

ramhit is a routine that stores a random number to a random location in memory.

stack_bomb is a routine that stores the value (frame+2) to a location 1/4 of the way up into the stack.

WORLD.C
---

This module contains only one line of modification. It now calls paged_data() before accessing segment data.

SECURE.C
---

This module contains 3 routines: init_security(), bomb(), and hit(). init_security() simply initializes some key memory locations for the whole security scheme and should be called once as part of the general system initializaton code. bomb() and hit() count-down their respective timers (if they are non-zero) and on a transition to zero they call stack_bomb() and ramhit() respectively. These two routines should be called in some regularly executed loop, or perhaps at vblank interrupt time.

HELPER.C
---

This module contains all the code to deal with the ASIC65. The WARRIOR security scheme ONLY USES 3 OF THESE ROUTINES DIRECTLY!!! They are: init_helper() (called in INT.ASM), HSP_INIT_SLOOP() (called by init_sloop() in INT.ASM), and HSP_GET_PAGE() (called by paged_data()). There is also a routine HSP_VERIFY that is #if'd out since it has been replaced by a macro (hsp_verify()) in HELPER.H. The other routines are largely holdovers from STEEL TALONS and can probably be removed, but I wouldn't bother if you don't need the ROM space.

HELPER.H
---

This module contains a number of macros and #defines used in HELPER.H
IT ALSO CONTAINS ONE VERY IMPORTANT MACRO WHICH THE SECURITY SCHEME IS DEPENDANT UPON!!! This is the hsp_verify() macro. This macro will cause the ASIC65 to return a 16-bit value and then subtract it from the data at location 0x78800 in the SLOOP area. It then stores this data in the location pointed to by one of the two parameters passed to the macro. If the value is zero, then all is well. If the value is non-zero, it means that the SLOOP is on a different page than the ASIC65 thinks it is on, and

therefore security has been breeched!! This information can be used in many ways which will be discussed later.

The other parameter that is passed to the macro (the first parameter in fact) should be some random 16-bit value that the macro uses to obscure the actual addresses that it accesses. How it does this is left as an exercise to the reader, but the upshot is that if you give it a different number each time you call it, then each call will produce sufficiently different code such that a pattern search of memory for the macro expansion would be futile.

Obviously, this header file will need to be #included into any module that will do a security check this way.

SECURE.H
---
This module contains 4 macros that may be used to implement security: sumit(), pgsum(), set_hit(), and set_bomb(). They are each intended to initiate "bad things". They are designed as macros so that they can be scattered all over and not just put in one spot. There are, however, slightly more devious ways to do the same things that pgsum(), set_hit(), and set_bomb() do, and these will be discussed later.

MAKEFILE
---
The changes to the MAKEFILE include adding the new modules as well as significantly modifying the memory map arrangement to conform to the separate document titled WARRIOR SECURITY SCHEME. This document also describes a general overview of the security scheme. The changes to the link ORDER and SECT commands are CRITICALLY IMPORTANT!!!!!

\ASIC65\WARRIOR: SLOOP1.ASM & ACE320.BIN
---
This module contains the ASIC65 code for the scurity scheme. There is only one thing that might need to be modified in this module. The table VER_TAB contains for 16-bit values that represent what's in the ROM at locations 0x78800, 0x7A800, 0x7C800, and 0x7E800. If after editing the world data and then using MY makefile, the data at these locations change from what is indicated in this file, the the file needs to be changed accordingly, and a make done in this directory. New ASIC65's should then be burned from the file ACE320.BIN. REMEMBER TO BLOW THE SECURITY FUSE!!

Dennis Harper can show you how to burn the ASIC65 chip.

\XH680X\MACRO.INC
---
This file contains some new macros for strobing the appropriate addresses on the SLOOP chip to exercise its various functions. It should be self explanatory. This file is needed for running the security system on a development station, primarily for the BKOFF macro which will disable the banking scheme thus allowing a successful download of the program to the SLOOP area (map as external static RAM).

IMPLEMENTING THE SCHEME
---

From this point, there are two MAIN parts to implementing the scheme.

First, you must merge my makefile with yours as outlined in the section above on the makefile. This will cause the proper data segments to be located in the appropriate places for the scheme to work. Also, at this time, check to see that the data at the "magic addresses" (0x78800...) have not changed, and if they have, be sure to modify the ASIC65 file SLOOP1.ASM as outlined above and burn a new chip. Install a SLOOP chip (Pat McCarthy can burn one for you) on your board. If you are working on a development system YOU MUST REMAP THE ADDRESS RANGE 0x78000 to 0x7FFFF to be target, and your PC board should have the static RAM mod for this memory area. BE SURE TO TURN OFF BANKING BEFORE DOWNLOADING THE MAIN PROGRAM!!!! The macro BKOFF has been defined for this purpose. At this point, the game will actually be using the SLOOP chip to do banking and will not work properly if either the ASIC65 or the SLOOP are not present or are faulty!

The second part requires installing the various verifies and booby-traps that have been prepared. init_security() must be called at system initialization. bomb() and hit() should be called at regularly intervals, perhaps every time through a main loop or every 8th VBLANK interrupt. You should then liberally sprinkle the code with calls to hsp_verify(). This is a macro in HELPER.H that checks to see that the SLOOP has the proper bank loaded. The macro stores the difference between what is expected and what is actually there into a location passed as a parameter to the macro. The macro should be placed in code that is executed only at very irregular intervals, thus making it not only more difficult to trace, but also delaying the affect. There are numerous ways to use the output of this macro. The most simple way is to test the value returned, and if not zero, invoke either the set_hit() or set_bomb() macro. This is fine except that it has the red flag of a compare instruction. Another way to use it is to pass the addresses of hit_timer or bomb_timer (see SECURE.C) to the hsp_verify() macro and let it set the timers directly. This disadvantage of this is that the values could be quite large causing a significant delay until the effect was noticed. You could also take the value and mask it to 5 or 6 bits before storing to those locations yourself. In any event, it would probably be wise to check that those locations are in fact zero already before either setting them or starting the whole verification process -- but there is the bloody compare again! It is important NOT to store to those addresses if they are already non-zero as that could potentially keep the bomb() and hit() routines from ever being able to count them down to zero.

Another way to the hsp-verify() macro is to call it in MODRIV at the place where you store to MO_TIMER. Currently you only call this once in a blue moon based upon frame. This is fine, and it would also be smart to only call hsp_verify() under the same circumstances. Then add one (1) to the result returned, and store that into MO_TIMER. A bad verify could cause the trap in Sam's chip to trip.

As a final measure, use either the pgsum/sum routines or the sumit macro to checksum sensitive parts of the security code. I would recommend checksumming the ramhit() and stack_bomb() routines. Also checksum the bomb() and hit() routines, and the place in the code that calls them!!! Obviously we could extend this concept ad nauseum and checksum the code that checksums other code but since the whole scheme is vulnerable to chip cloning anyhow, why bother! Remember that changing ANY code after calculating what the checksums should be will likely cause the code being checksummed to move, and thus change the checksums!! USER BEWARE!!!!

FINAL TOUCHES AND EXTRA CREDIT
---

I would strongly reccomend that when the changes to the segment data has been finalized, that you do two additional things. First, check the link map to determine the size of the non-banked segments. My makefile currently has them located at 0x70000. I would recommend moving up so that they don't leave a large gap between themselves and the "sloop" section. This will keep the sloop code from hanging out in an isolated island of code. DO NOT, however, push the segment right up against the bottom of the soop section. Leave a 4 or 5 byte gap so that there can be no chance of an address overrun hitting the FREEZE_OFF address located at the start of the sloop section.

Secondly, it would be a good idea to scramble bits A14 & A13 of those addresses in the array "data_ptr" (in WORLD.C) that are actually in the SLOOP area. Since all pages have the current page as data when banking is active (the normal game mode), it doesn't matter which bank the data is read from. The problem with this is that it is unlikely that the Microtech linker will do this math for you, so you would have to wait until EVERYTHING is finalized and look up the addresses in the link map and enter them as raw hex values in the table by hand. This would make it a little harder for a hacker to simple disable the SLOOP turning the SLOOP area into non-banked memory again.

One final possibility would be to put more data into the SLOOP area, but that gets very involved. If you do decide to do it it would require that you:

1) Extend the lookup table in ASIC65 (SLOOP1.ASM) and reburn the chip.
2) Add appropriate enums to the DATA_SEG enum in WORLD.H.
3) Do a paged_data() call before trying to access this data and then get the address by indexing the "data_ptr" array in WORLD.C.
4) BE SURE TO RESTORE PAGE OF THE CURRENT SEGMENT BEFORE CONTINUING!! This is simply done by doing the following:

```
paged_data(seg);
```

And it will insure that the current segment's seg_data table is available in the resident page.

LASTLY...
---

I hope I have covered everything, and this all goes smoothly. It is quite possible that I have not, and if you have any trouble, feel free to contact me (but not the week of March 21-28 -- I'll be out of the country!!) and I'll make arrangements to help you out.

Most of all, good luck with the project and I hope you make a bundle of dough for yourself and ATARI!!

- The Gonz -

# Security

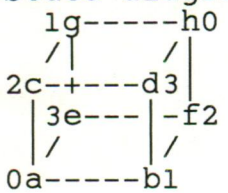
1) Bring up your system with our latest
2) unget makefile
2.5) install helper for Andy & Gary
3) put everything else (source & include) on Gary's system
4) Merge & Add scroll segs to makefile
5) install security hardware
6) verify all works

Int.asm
sloop.c (ASM)
makefile
world.c
scroll.asm
helper.c
helper.h
secret.asm
secure.c
secure.h

Confidential
1/9/92
Pat Mc Carthy
**********

SL2 state machine.

State diagram




Modifier registers
Freeze
Bound
Pull_enabl

Bank_enable
Bank

MAGIC words.
---
Ku, Kd, Kl, Kr, Kf, Kb, Freeze_on, Freeze_off, Pull_enab, Pull,
Bound_on, Bound_off, Bank_on, Bank_off

The CUBE and cube transition addresses.
---
The scheme is basicaly an 8 state CUBE shell game, as shown above.
The 8 states are labeled a-h. The number beside each state label is
the bank that is associated with that state.
Move around the CUBE by using cube-transition address keys :
Ku (up), Kd (down), Kl (left), Kr (right), Kf (forwards) and Kb (backwards).
For example to get from state 'a' to 'b' use Kr.
To get from state 'b' to 'a' use Kl.
To get from state 'a' to 'g' use Ku and Kb.

The FREEZE register.
---
When the 'Freeze' register is set (with 'Freeze_on') the CUBE state
is frozen. This locks the current state regardless of the application
of cube-transition keys. Use the 'Freeze_off' address key to enable
the cube-transition keys again.

The PULL address.
---
To pull a bank, first enable pulls with the 'Pull_enab' key
address. Then 'Pull'. The number corresponding to the state
will be transferred to the 'Bank' register. Pull only works when
'Pull_enabl' is on. 'Pull_enabl' is turned off automatically by
'Pull' and also by 'Freeze_off'.

The BOUND register.
---
The 'Bound' register limits transitions to within the CUBE when it is
set (with 'Bound_on'). For example, if the current state is 'b' and Kr
is used the new state will still be 'b'. When the 'Bound' register is
cleared (with 'Bound_off') the cube-transitions will wrap around. For
example, if the current state is 'b' and Kr is used the new state will
be 'a'.

The BANK_ENABLE register and mux.

---

The cpu address appears on the output pins except when the banked address space is decoded and the 'Bank_enable' register is set. Use 'Bank_on' and 'Bank_off' to set and clear this register.

## Initialisation state.

---

The chip powers up in the following state :

```
CUBE state  : 'a'
Freeze      : off
Pull_enabl  : off
Bound       : on
Bank_enable : off
Bank        :  0
```

There is no reset key. However, the cube can be initialised from an unknown state by setting Freeze_off, Bound_on and executing three cube-transition keys, one for each direction (ud, lr, fb).

## Additional features.

---

Multiple keys: Multiple keys can be added easily. For example, you might want 4 'Pull's or 2 'Ku' keys or 3 'Bound_on' keys.

State specific keys : There are 8 state transitions for each transition key. For example, Ku is used for state transitions a-c, b-d, e-g, f-h, and also the 4 wrap around transitions c-a, d-b, g-e and f-h. Keys that are specific to only some of those transitions are possible.

Combination keys : There are no combination keys (keys that are required consecutively) in this implementation. They could be added. In particular, 'Freeze_on' and 'Freeze_off' are gateway keys that may warrant more complexity.

A lockout key sequence : A sequence that when triggered prevents the chip from doing anything at all until the chip is powered down and up again. Even resetting the game would not reset it. You would use this as a first level of defence when you detect tampered code.

A second game : ? is it a good idea.