SADE 1.4a3
Release Notes

This version of SADE introduces a new feature, memory protection. These release notes describe how to use memory protection in SADE and provide a list of potential pitfalls when using memory protection. For general information on using SADE, please refer to the SADE 1.3 Manual and SADE 1.3.3 Release Notes.

△ **Warning** This feature can cause disastrous results if you're not careful, however, it can greatly assist you in isolating memory stomping bugs. Please refer to the section on *Potential Pitfalls* for options to avoid. △

Changes Since SADE 1.4a2

- SADE would occasionally fail while defining a certain number of user strings on systems with greater than 16MB of RAM. This has now been fixed.
- Under certain low memory conditions, SADE may not have been able to retrieve correct type information. This has now been fixed.

Changes Since SADE 1.4a1

- SADE was handling trace exceptions generated by MacsBug. While running SADE in the background, if you attempted to set an address break from MacsBug and then execute "go", SADE would take control. This has now been fixed.

Using Memory Protection

- The memory protection feature requires a Macintosh with an MMU (Mac II with 68851 PMMU, or any 68030 or 68040 Macintosh). It also requires that VM be enabled.
- The feature is used like any other command in SADE; execute it from the worksheet or from a script.

- The memory protection feature will not work under A/UX because it requires direct access to the processor's MMU hardware. In UNIX, only kernel code is privileged to do this.

Syntax

`protect [from  target | all]` <address expression> [<number of bytes>]

`unprotect all |` <address expression> [<number of bytes>]

`list protect`

`from target`	Restrict SADE to reporting only those write protection faults which occurred when the program counter was within the application's heap zone. This is the default.
`from all`	SADE will report write protection faults regardless of where the program counter was at the time of the write. Refer to the section on potential pitfalls for a complete description of this option.
<address expression>	any expression which SADE can evaluate as an address, e.g. '&foo', '$1C48D0A', 'EVENTS.MAINEVENTLOOP.(3)', etc.
<number of bytes>	an optional value specifying how many bytes to protect beginning at that address. The default value for <number of bytes> is 4.
`list protect`	Lists all currently protected addresses.

Description

- Protecting a region of memory causes SADE to detect any writes to that region and report them, e.g.,
  ```
  Write to protected memory detected at DRAWROTATINGDIAL.(3)+$0002
  ```
- When the exception is reported, the contents of the protected memory will have been changed.
- Because 68030 and 68040 processors are pipelined and bus error exceptions are imprecise, the PC at the time the exception is detected may be several instructions past the one which actually caused the fault. Usually this means SADE will stop at the next source statement, but occasionally it will stop far from the faulting instruction, e.g. when the faulting instruction is followed by a branch or subroutine call.

- *Note:* All protected addresses are unprotected when you kill or untarget a target or quit SADE.

- Memory protection applies only to absolute addresses and should not be used to protect relocatable data, such as data in unlocked handles.

Examples

- The following example describes a typical programming error which would occur if you define an n-element array in C and try to write to element[n] of that array:

```
main ()
{
      int myArray[5] = {10, 20, 30, 40, 50};
      int int1 = 1200;
      int int2 = 1400;
      ...
      myArray [5] = int2; /* fault here, int1 should change */
}
```

  In the previous example, the value `int1` would change instead of the last element of the array, as the programmer probably intended. To determine the location of the write to `int1`, you could use the protect feature as follows:

```
protect &int1
```

  SADE responds with the following confirmation:

```
Protect 4 byte(s) at $00EF753C  (from target code only)
```

  You may also verify for yourself which addresses have been protected

```
list protect
$00EF753C .. $00EF753F  (from target code only)
```

  When you execute the program, the write protection fault will occur and SADE will suspend the program at (or just after) the instruction which caused the error:

```
Write to protected memory detected at main.(4)
```

  You may now remove the protection by either killing the target or executing:

```
unprotect all
```

- You can protect code as well as data. This is especially useful for catching memory stompers which overwrite target code and cause mysterious crashes. For example, to protect a function called `DoUpdate` which is followed by another function `DoActivate`, use:

```
protect DoUpdate DoActivate - DoUpdate
```

- For a stack sniffer which is 100% guaranteed (unlike the VBL-based OS routine), protect the address in `ApplLimit`:

```
protect ApplLimit
```

- SADE's sizeof function is handy for specifying a byte count, e.g. to protect variable `foo` use:

```
protect &foo sizeof(foo)
```

Potential Pitfalls

When using “protect from all”

- This is the most dangerous option and should be used with **extreme caution** . When this option is used, SADE will attempt to report memory protection faults regardless of the location of the program counter. If you protect memory outside the target's parti - tion (generally, the area bounded by the addresses in ApplZone and A5) and the system writes to that location, your machine may crash. This is because SADE has no knowl - edge of who has legitimate reasons to write to particular addresses, and if SADE inter- rupts non-reentrant system code (i.e. most system code) it will not be possible to switch SADE in and your machine will hang. Avoid the temptation to protect location `$0` or any other part of low memory except in “protect from target” mode.
- For similar reasons, it is usually a bad idea to try to protect addresses in the target’s stack. For example, if you protect a local variable which then goes out of scope before you unprotect it, you will very likely crash because parts of both SADE and the Process Manager also run on the target's stack and they are bound to hit the protected address.
- Don’t set breakpoints in code which has been protected. SADE must write to code to set breakpoints.
- Don't protect variables which may be changed by the system. For instance, if you write a Pascal program and protect the variable &mouse and your code makes a trap call to `GetMouse(mouse)`, SADE will suspend execution of the system code and you'll no longer be able to retrieve symbol information for the source. You should kill the target if this occurs. Continuing to step or execute the program may result in a system crash.

When using either “protect from all” or “protect from target”

- Don’t protect very high address ranges (or protect a negative number, like -5). This is likely to freeze your system.
- Be sure to protect local variables only after they have been defined, (after the LINK A6 instruction has been executed). Otherwise, the address you protect will not be valid.

- If you protect local variables, be sure to unprotect them before they go out of scope. If you don't, you'll be protecting an address that will be used by someone else. Remember that locals to `main()` are still locals.
- Don’t attempt to protect variables which are stored in registers. If you do, you will get the following error message:

  ```
  ### Cannot take the address of a register
  ```

  In addition, SADE will probably execute very slowly due to all the writes to the low memory location you just accidentally protected. Be sure to unprotect the variable if this occurs.

  If you protect `ApplLimit` because you suspect your stack is overwriting the heap, once you encounter the fault do not continue to execute your program. The exception handling code that SADE executes pushes parameters on the user's stack which will corrupt the heap.